MASTER EN SCIENCES INFORMATIQUES

L'utilisation d'un outil d'analyse statique de code permet-il à des novices d'améliorer leur apprentissage de la programmation ?

Bertrand, Vincent

Award date:
2017

Awarding institution:
Universite de Namur

Université de Namur
Faculté d'informatique
Année académique 2016–2017

L'utilisation d'un outil d'analyse statique de code permet-il à des novices d'améliorer leur apprentissage de la programmation ?

Vincent BERTRAND

Promoteur : _____(Signature pour approbation du dépôt - REE art. 40)
Benoit FRÉNAY

Co-promoteur : Julie HENRY

Confidentialité du mémoire : publique

Mémoire présenté en vue de l'obtention du grade de
Master en Sciences Informatiques.

Résumé

Ce mémoire évalue l'utilité qu'un outil d'analyse statique de code peut apporter à des novices en programmation dans le cadre de leur apprentissage de cette matière. Pour y parvenir, un outil est sélectionné, SonarQube, et deux expériences sont construites. Le public cible de celles-ci sont des étudiants en première année de bachelier en Sciences Informatiques et en Ingénieur de Gestion à finalité Management de l'Information à l'Université de Namur. Dans un premier temps, l'objectif est de déterminer si l'outil apporte une plus-value aux étudiants dans le cadre de leur apprentissage. Pour cela, une sensibilisation à la qualité logicielle leur est proposée au travers d'un premier contact. Durant cette expérience, plusieurs types de données sont récoltés afin de s'assurer de la complétude de l'analyse : enregistrement vidéo de l'expérience, questionnaire et résultats d'analyse du code des étudiants réalisé par SonarQube. Dans un second temps, l'outil est proposé à l'utilisation pendant une période de six semaines afin de déterminer si SonarQube peut être utilisé en autonomie par les étudiants. Pour leur offrir un maximum d'autonomie, un environnement personnalisé permettant d'abstraire la partie technique est mis en place. Les deux expériences ont rempli leurs objectifs. La première a produit des résultats encourageants quand, pour la deuxième, ils ont été un peu décevants.

Mots Clés

Enseignement de la programmation, apprentissage de la programmation, analyse statique de code, sensibilisation à la qualité logicielle, SonarQube

Abstract

This thesis evaluates the utility that a static code analyser can bring to novices in computer programming during their learning phase. The target public are students in first year in Computer Science and students in first year in Management Engineer option Information Management at Namur University. To be able to validate this hypothesis, a tool is selected and two experiences are built. Firstly, the goal is to determine if the tool provide an added-value to students during their learning phase. To do so, students were proposed to use a professional analysis tool to raise their awareness to software quality. During this first experience, many types of data are gathered: video records, questionnaires and students code analysis results. Secondly, the tool is proposed to be used during six weeks by students on their own. To let them become autonomous, a custom environment is built to abstract the technical aspects. The two experiences reach their goals. The first produces some promising results when for the second, it was a little disappointing.

Keywords

Computer programming teaching, computer programming learning, static code analysis, awareness to the software quality, SonarQube

Merci !

À *Monsieur Benoît FRÉNAY*, pour avoir promu ce mémoire, pour le temps accordé lors des multiples réunions, pour la mise à disposition d'heures de cours avec ses étudiants permettant la réalisation des expériences décrites dans ce travail et *Madame Julie HENRY* pour avoir co-promu ce mémoire, pour ses conseils concernant l'aspect analytique, pour sa grande aide, ses lectures en profondeurs et corrections apportées durant la rédaction du présent document.

Merci !

Aux Étudiants de première année de *Bachelier en Sciences Informatiques* et aux Étudiants de première année de *Bachelier en Ingénieur de Gestion filière Management de l'Information* de l'année scolaire 2016 – 2017 pour avoir participé aux expériences décrites dans ce mémoire.

Merci !

À *Monsieur Carlos RUIZ SANCHEZ* pour avoir réalisé le déploiement sur le réseau de l'Université de Namur d'applications utiles aux expériences décrites dans ce mémoire.

Merci !

À *Monsieur Benoît VANDEROSE* pour avoir transmis des informations utiles sur la Qualité Logicielle.

Merci !

À *Madame Delphine TURCHETTI* pour avoir transmis des informations mathématiques utiles à la construction et l'analyse de graphiques.

Merci !

À *Madame Aude NGUYEN*, sans qui ce mémoire n'aurait pas existé, pour m'avoir mis en relation avec les promoteurs.

Merci !

Á toutes les personnes non-citées ayant aidé de près ou de loin à la réalisation de ce mémoire.

Merci infiniment à tous !

Table des matières

Introduction

L'informatique prend une place de plus en plus importante dans notre vie : au niveau industriel, elle remplace des ouvriers ; au niveau économique, elle assiste des employés ; au niveau de la vie quotidienne, elle amuse des aficionados, etc. Bref, l'informatique est partout et le sera encore plus demain. La cadence actuelle de formation ne parvient pas à pourvoir aux postes vacants. Il est absolument nécessaire de pallier à ce problème.

Une solution logique existe, elle se compose de deux étapes : premièrement, attirer plus d'étudiants à choisir l'informatique comme formation et deuxièmement les garder et les amener à terminer cette formation. Les deux étapes sont nécessaires car lorsque les étudiants ne se sentent pas à l'aise dans leur matière, même s'ils y voient de l'intérêt, ils ont tendance à changer d'orientation ou à abandonner. Donc la seule solution d'en recruter plus n'est pas suffisante si l'enseignement ne devient pas plus efficace dans le même temps. Il est donc nécessaire d'améliorer encore l'enseignement (l'apprentissage ?) de la programmation.

Les enseignants cherchent sans relâche des moyens d'améliorer leur cours, de les rendre plus attractifs, etc. Il s'agit très souvent de solutions générales, qui sont appliquées pour tous les étudiants sans discrimination, donc sans tenir compte de leurs forces et faiblesses. Il serait dès lors intéressant de trouver un moyen de cibler celles-ci afin de leur offrir un apprentissage sinon personnalisé du moins assisté. Guidés par un ensemble d'indications leur permettant de pointer les notions moins acquises, les étudiants seraient alors en mesure de renforcer leur apprentissage afin d'améliorer leurs compétences.

Des outils répondant aux critères décrits ci-dessus existent pour les professionnels : les outils d'analyse statiques. Il s'agit de logiciels conçus pour analyser le code d'un projet de manière statique (i.e.: sans exécution) pour en déduire des informations permettant de définir la qualité du code. Ces informations peuvent être l'indication du non-respect d'une règle[1], le nombre de lignes de code utiles (hors commentaires, lignes vides, etc.), le degré de complexité d'une fonction / méthode ou d'une classe, etc. Toute une série d'informations qui pourraient indiquer à l'étudiant que le programme qu'il produit est « *bon* » ou non et en quoi il ne l'est pas le cas échéant.

C'est ici que le mémoire intervient. Il permettra de valider l'utilité de ce type d'outils dans le cadre de l'apprentissage de la programmation en donnant des éléments de réponse à la question : « *L'utilisation d'un outil d'analyse statique de code permet-il à des novices d'améliorer leur apprentissage de la programmation ?* ». Ce travail va être guidé par la validation de deux hypothèses préliminaires. Premièrement, il est nécessaire de s'assurer que ce type d'outil apporte une plus-value dans le cadre de

[1] Fusionner deux « if » successifs.

l'apprentissage. Deuxièmement, l'outil qui sera sélectionné pour ce mémoire se doit d'être utilisable par les étudiants, en autonomie.

Pour pouvoir répondre à ces hypothèses, le contact avec des étudiants est obligatoire. Le public visé sera donc composé d'étudiants de l'Université de Namur inscrits en première année d'enseignement du cursus « Master en Sciences Informatiques » et « Ingénieur de Gestion filière Management de l'Information ».

La première des deux expériences leur offrira un premier contact avec l'outil d'analyse statique choisi dans le cadre de ce mémoire. Étant organisés en groupe pour la réalisation d'un mini-projet, celui-ci prenant place dans leur cours de programmation, à un moment donné, leur code sera analysé et il leur sera demandé de parcourir, en présence du chercheur, le résultat de cette analyse. Le but ici est de déterminer dans quelle mesure une simple sensibilisation à la qualité logicielle peut avoir un impact sur leur manière de travailler. Comprennent-ils les informations affichées par l'outil ? Sont-ils prêts à utiliser l'outil ? Vont-ils corriger les erreurs soulevées par l'outil ? etc. sont toutes des questions auxquelles des éléments de réponse sont susceptibles d'être trouvés.

Après configuration, l'outil sera mis une deuxième fois à disposition des étudiants pour une utilisation en autonomie. Ils seront donc seuls, sans accompagnant, lorsqu'il s'agira d'interpréter les résultats d'analyse. L'expérience trouve plusieurs utilités : la validation de la configuration de l'outil, son succès auprès des étudiants, l'identification d'éventuelles problèmes que les étudiants pourraient éprouver. Le choix des règles à suivre est-il pertinent ? L'interface convient-elle pour des programmeurs novices ? À quelle fréquence les étudiants réalisent-ils une analyse ? etc. sont des questions auxquelles l'expérience devrait apporter une réponse.

La rédaction du mémoire est organisée en huit chapitres. Sa logique suit la démarche mise en place tout au long de l'année.

Tout d'abord, un état de l'art est réalisé dans le **premier chapitre** afin de faire le point sur les éléments « théoriques » importants de ce mémoire : l'apprentissage de la programmation et la qualité logicielle / analyse statique à l'aide d'outils spécifiques.

Ensuite, après avoir abordé des situations générales dans l'état de l'art, il s'agit de les contextualiser dans le cadre de l'Université de Namur. Le **deuxième chapitre** se concentre donc sur la description du contexte et des besoins qui ont amené la problématique.

Le **troisième chapitre** consiste en la clarification de la question de recherche et des hypothèses de recherche qui guident l'ensemble du travail.

Le **quatrième chapitre** est plus technique et présente l'outil d'analyse statique qui a été choisi pour la réalisation de ce mémoire. Ce chapitre est structuré en deux parties :

en premier lieu, il est nécessaire de clarifier les raisons qui ont poussé à choisir cet outil et pas un autre ; en deuxième lieu, ce dernier est présenté de manière fonctionnelle, mais également de manière technique.

L'ensemble des éléments de base étant décrit, le **cinquième chapitre** résume la méthodologie de recherche mise en place. Il consiste en la description des expériences, des résultats récoltés durant celles-ci et de la manière dont ces résultats ont été traités.

Le **sixième chapitre** se concentre sur la première expérience et se partage en deux parties. Premièrement, l'ensemble des données y sont décrites : questionnaire de satisfaction d'après séance, résultats des analyses de code des étudiants et enregistrements vidéo. Deuxièmement, les données sont analysées pour pouvoir en tirer les conclusions nécessaires à la suite du travail.

Dans le **septième chapitre,** le travail de préparation nécessaire à la mise en place de la seconde expérience est présenté. En effet, afin d'offrir aux étudiants l'autonomie désirée face à l'outil d'analyse statique, il a été nécessaire d'en abstraire son aspect technique. Ceci s'est fait au travers de la conception d'un outil de soumission sui se veut simple d'utilisation. Le chapitre est structuré en deux sections. La première sert à détailler le développement de cet outil de soumission. La deuxième aborde la configuration de l'outil d'analyse statique sélectionné qui a été utilisée durant la deuxième expérience. Celle-ci est elle-même divisée selon deux aspects : d'abord, il est expliqué comment, au travers de son interface, un administrateur doit gérer des utilisateurs et les permissions associées et ensuite, la configuration d'un profil de qualité (ensemble de règles à suivre) est exposée.

Le **huitième chapitre** décrit les données récoltées durant la deuxième expérience et les analyse ensuite.

Le **neuvième** et avant-dernier **chapitre** permet de tirer les conclusions des deux expériences réalisées. S'en suit une section dédiée aux suggestions permettant de guider d'éventuels travaux désirant approfondir la problématique développée durant ce mémoire.

Enfin, le **dernier chapitre** termine ce travail en concluant sur la réponse à la question de recherche développée dans le troisième chapitre.

Chapitre 1 : État de l'art

1.1. Apprentissage de la programmation

Il n'est pas rare d'entendre dans les média que le métier de développeur est un « métier en pénurie », un « métier d'avenir » ou un « métier ayant énormément de débouchés »[2]. Cette tendance s'accentue par le nombre limité d'étudiants faisant ce choix d'études (Conseil Wallon de la Politique Scientifique, 2013). De plus, ceux n'ayant pas développé suffisamment les compétences requises ont tendance à jeter l'éponge plutôt que de persévérer dans l'acquisition de celles-ci (Gomes & Mendes, 2007a). Il est donc très important de trouver un moyen de donner, laisser, rendre le goût de la programmation aux étudiants.

Cette tendance n'est pas aidée par les performances des étudiants en programmation qui se révèlent, sous certains aspects, moins bonnes qu'espérées (McCracken, et al., 2001). En effet, les examinateurs ont mis en place ont mis en place une étude internationale afin de répondre à la question « *Do students in introductory computing courses know how to program at the expected skill level* ? ». La réponse obtenue est sans conteste : « Non ». Plusieurs des solutions comportaient des erreurs de syntaxe, certains étudiants n'ont pas réussi à décomposer le problème, d'autres n'avaient même pas testés leur solution, etc. Les auteurs pointent un élément important à prendre en compte : beaucoup d'étudiants ne sont pas conscients de leurs propres difficultés car se concentrent sur l'apprentissage de la syntaxe.

D'autres études ont également pointé des difficultés rencontrées par les étudiants : les problèmes d'apprentissages de syntaxe (Gomes & Mendes, 2007b), les problèmes demandant une compréhension et une connaissance générale trop importante du programme à concevoir (Lahtinen, Ala-Mutka, & Järvinen, 2005), la faiblesse dans la capacité à résoudre des problèmes (Gomes & Mendes, 2007a), etc. Ceci souligne les difficultés rencontrées durant l'apprentissage de la programmation. Concernant les problèmes d'apprentissage de la syntaxe d'un langage, proposer des messages plus compréhensibles n'aide pas toujours. En effet, les étudiants ont néanmoins besoin de compiler autant de fois le code que des étudiants ne bénéficiant pas de ces messages améliorés (Denny, Luxton-Reilly, & Carpenter, 2014) (Denny, Luxton-Reily, Tempero, & Hendrickx, 2011). Ces conclusions doivent cependant être tirées avec un peu de hauteur puisqu'une des erreurs de syntaxe la plus courante concerne l'absence de point-virgule en fin d'instruction.

Dans le but de leur faciliter cette période d'apprentissage, certains ont tenté d'améliorer les messages d'erreurs venant des compilateurs (java, entre autres). Ceci a eu pour effet de diminuer le nombre d'erreurs produites par des étudiants (Becker, 2016). D'autres ont spécifiquement conçus des systèmes pour les débutants. Pour cela,

[2] https://www.leforem.be/MungoBlobs/221/484/Metiers_penurie_2017_2018_V_04_07.pdf

il est abstrait une partie rébarbative de l'apprentissage de la programmation (absence de syntaxe, langage plus proche du parlé, etc.), ce qui permet aux apprenants à se concentrer sur la logique, sans élément perturbateur (Kelleher & Pausch, 2003). Ces outils sont, de manière générale, destinés à des enfants ; Scratch[3] est un des outils les plus connu et l'âge le plus représenté est 12 ans avec 2 102 684 inscrits[4]. Leur but est, entre autres, d'empêcher les erreurs de syntaxe de manière ludique (Maloney, Resnick, Rusk, Silverman, & Eastmond, 2010). L'ensemble des solutions décrites ci-dessus tentent de trouver de répondre à des problèmes généraux.

1.2. La notion de Qualité Logicielle

Il existe différentes manières de diagnostiquer ses propres faiblesses d'apprentissages, telles que les évaluations formatives. Des études ont mis en évidence l'utilisation de métriques pour le cas spécifique de la programmation. En effet, il est montré que l'utilisation de ce genre d'informations peut aider à identifier les faiblesses dans les programmes évalués (Kasto & Whalley, 2013). Cela revient à mesurer la qualité globale d'un projet / exercice.

La notion de qualité logicielle peut être arbitraire, elle a donc été normalisée / modélisée au travers de différents modèles : McCall, Boehm, Dromey, FURPS, ISO, etc. Une de ces normes les plus connues est la norme ISO9126 (cfr. Figure 1), qui définit la qualité d'un logiciel à partir de six catégories :

1) Le fonctionnement représente la conformité qu'à le logiciel par rapport aux spécifications données et comprend l'adéquation, la précision, l'interopérabilité et la sécurité ;
2) La fiabilité représente l'exactitude des données peu importe les conditions d'utilisation et comprend la maturité du projet, la tolérance aux pannes et la capacité d'un logiciel à recouvrer la santé après une panne ;
3) L'utilisabilité représente le degré de difficulté dans l'apprentissage de l'utilisation et comprend la compréhension, l'apprentissage, l'opérabilité, l'attrait visuel ;
4) L'efficacité représente la performance du logiciel et comprend le comportement dans le temps et l'utilisation des ressources ;
5) La maintenabilité représente la capacité à être maintenu, corrigé, adapté et comprend la facilité d'analyse, la difficulté de changement, la stabilité et la testabilité ;

[3] https://scratch.mit.edu/

[4] https://scratch.mit.edu/statistics/

6) La portabilité représente le degré de fonctionnement d'un logiciel dans un environnement différent de celui pour lequel il a été développé et comprend l'adaptation, l'instabilité, la coexistence et le caractère remplaçable du logiciel.

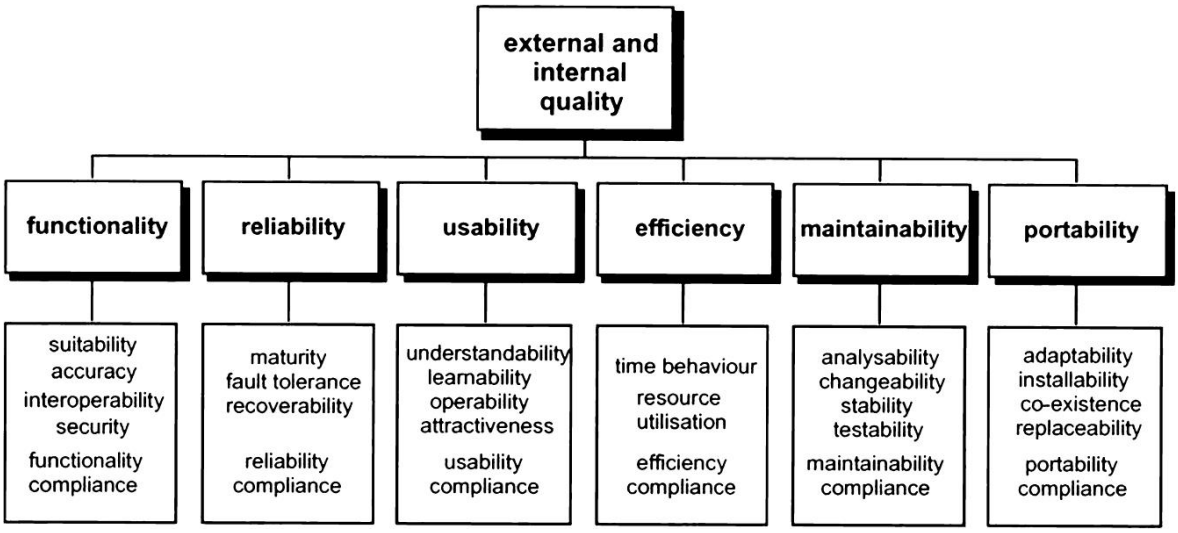


Figure 1: Attributs définis dans la norme ISO 9126 (Image tirée du cours IHDCM031 – Partie 5 – slide 33).

Deux axes principaux permettent d'assurer qu'un logiciel est de qualité : la vérification / validation et la mise en place d'un plan de qualité logicielle. D'une part il s'agit de s'assurer que le programme fait ce qui est attendu de lui et continue de fonctionner de manière correcte même si l'utilisation diverge des cas normaux. C'est donc très utile mais ne permet pas à des étudiants de prendre connaissance de leurs faiblesses. Le second axe quant à lui permet de s'assurer que l'équipe de développement met en place et respecte les standards de qualité définit lors de la conception d'un logiciel. Il peut s'agir de simples conventions de nommage, de contraintes sur l'architecture, de respect de métriques, etc. (Unamur - IHDCM031, 2015 - 2016).

Il existe bon nombre de logiciels permettant d'analyser le code source afin de faire ressortir d'éventuels problèmes dans l'écriture de celui-ci, de calculer ses métriques, etc. (Ataíde, 2014). Qu'ils soient gratuits ou payants, intégrés à un IDE ou non, ils poursuivent tous le même but : aider les développeurs dans leur travail en soulignant les faiblesses de leur code.

Chapitre 2 : Contexte et problématique

Ce mémoire a pour public cible des étudiants de l'Université de Namur (UNamur) en première année de bachelier en Sciences Informatiques (Info) et en première année de bachelier en Ingénieur de Gestion filière Management de l'Information (IngMI). Dans la suite de ce travail, « les étudiants » fera référence aux étudiants décrits dans ce paragraphe.

Ces cursus ne sont pas des plus populaires dans le catalogue de formation que propose l'Université de Namur : l'année scolaire 2015 – 2016 a bénéficié de 134 inscrits cumulés (71 en INGMI et 63 en INFO) quand la suivante, 2016 – 2017, n'en a bénéficié que de 116 au total (55 en INGMI et 61 en INFO).

2.1. L'apprentissage de la programmation

Les deux formations précédemment citées ont en commun des cours d'apprentissage de la programmation : « Introduction à la programmation » (INFOB131) et « Laboratoire de développement de programme » (INFOB132). Le premier des deux a pour objectif principal de permettre à des étudiants, néophytes ou non, d'apprendre les rudiments de la programmation tels que « *manipuler des variables et des valeurs* », « *abstraire du code à l'aide de fonction* » [5], entre autres. En d'autres mots, à la fin du cours, les étudiants doivent être capables de « *résoudre des problèmes simples à l'aide d'algorithmes adaptés…* »[1]. Le deuxième, quant à lui, vise a leur apprendre à gérer un projet de développement complet.

Le cours INFOB131 se déroule lors du premier semestre. Il est structuré en trois blocs, ponctués chacun d'un mini-projet permettant de mettre les acquis en pratique. Pour les réaliser, les étudiants sont organisés en groupe de six membres et ont à leur disposition un environnement de développement gratuit : Enthough Canopy[6]. Le langage utilisé pour développer les programmes est le Python.

Le « Laboratoire de développement de programme » trouve sa place dans le second semestre. Il permet aux étudiants de mettre en pratique les acquis du premier semestre, notamment les notions de programmation enseignées durant le cours INFOB131. Il est structuré selon plusieurs échéances claires que les étudiants doivent respecter afin de produire des petits livrables et de terminer par les assembler de manière cohérente.

Par l'intermédiaire de ces deux cours, les étudiants apprennent ce qu'est la programmation et comment la pratiquer, ainsi que quelques bonnes habitudes telles

---

[5] https://directory.unamur.be/teaching/courses/INFOB131/2017

[6] https://www.enthought.com/products/canopy/

que la factorisation de code en fonctions, la rédaction de spécifications techniques, la mise en place de commentaires utiles, etc.

```
1   def test_case(board, player_id, i):
2       """Test if a case contains  ship or not.
3
4       Parameters
5       ----------
6
7       board = board playable where cases are tested.
8       player_id = player's id (str)
9       i = case wanted for the location of a ship and that serves to test the other cases.(int)

10      Raises
11      ------
12
13      ValueError : wrong player_id
14
15      returns
16      -------
17
18      False if a ship is located around the cases tested.
19
20      """
21      if player_id == 1 or player_id == 2:
22          test = (-10, -11 , -9, -1, 0, 1, 11, 10, 9)
23          for k in test:
24              l = i+k
25              if l not in board:
26                  print ''
27              else:
28                  if player_id == 1:
29                      if board[i+k]['statutP1']== '#':
30                          return False
31                  elif player_id == 2:
32                      if board[i+k]['statutP2']== '#':
33                          return False
34      else:
35          raise ValueError('Wrong player_id')
```

Code 1: Exemple de fonction trop profonde (provient d'un étudiant en BAC1 INFO – Année 2015 – 2016).

Ces bonnes habitudes ne sont malheureusement pas adoptées équitablement par tous. En effet, il n'est pas rare de constater des fonctions composées d'un nombre jugé trop important d'instructions (compte tenu du problème posé à l'étudiant), trop complexes (du point de vue algorithmique) ou possédant une profondeur trop importante (imbrication de « if », « else », « for », etc.). Cela met en évidence sans conteste un déficit de factorisation. À titre d'exemple, le Code 1 illustre un problème de profondeur de code trop importante. Si, sur 15 lignes de code, la fonction est correctement définie avec rédaction de spécifications, elle présente néanmoins pas

moins de cinq niveaux : lignes 21, 23, 27, 28 et 31, 29 et 32. Un deuxième exemple, illustrant une fonction trop complexe, est consultable en Annexe 1. Celle-ci est partagée en deux traitements différents : le premier, de la ligne 9 à la ligne 22, pourrait être abstrait en une fonction externe; le deuxième, de la ligne 24 à la ligne 422, semble contenir des répétitions et pourrait, avec un peu de travail, être factorisé de manière plus efficace.

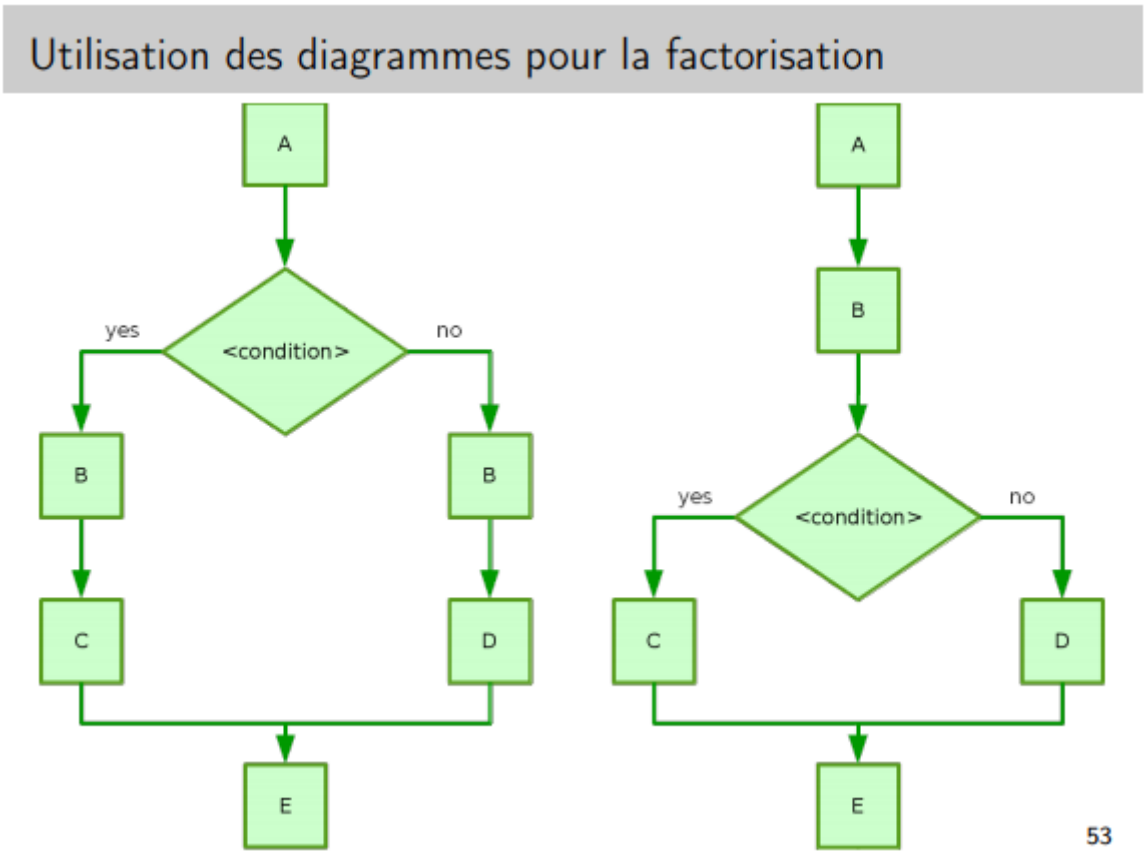


Figure 2: Exemple de diagramme utilisé pour aider la compréhension de la factorisation (Image tirée du cours INFOB131 – Cours 3 – Slide 53).

Les exemples cités ci-dessus sont issus de programmes réalisés par les étudiants de l'année scolaire 2015 – 2016 durant le « Laboratoire de développement de programme ». Ils témoignent donc d'un niveau moyen d'apprentissage atteint par des étudiants en fin de première année de bachelier. Ces constatations renforcent l'impression que la matière n'est pas suffisamment acquise. Impression ressentie suite à l'observation des résultats des étudiants décrits au début du chapitre.

Afin d'aider les étudiants, les professeurs cherchent des pistes didactiques à mettre en place. Par exemple : Monsieur Frénay a introduit l'utilisation de diagrammes (cfr. Figure 2) pour illustrer visuellement la structure du programme tels que les branchements conditionnels, les boucles, les appels de fonctions, etc. D'un point de vue pratique, les étudiants ont donc la possibilité d'utiliser cette méthode jusqu'à ce qu'ils maitrisent suffisamment les notions.

Durant la première année de bachelier, trois sessions d'examens sont offertes aux étudiants et la Figure 3 et la Figure 4 illustrent les taux de réussites cumulés du cours INFOB131 par rapport au nombre d'étudiants inscrits durant ces différentes sessions[7]. Il est intéressant de remarquer que, en fin de troisième session de l'année 2015 - 2016, environ 80% des étudiants en INFO inscrits ont réussi quand le taux correspondant aux INGMI s'approche seulement des 65 %. Ces taux peuvent être satisfaisants, ceux de la première session le sont nettement moins. En effet, les étudiants en INFO ne sont qu'environ 55% à réussir l'examen en première session et à peine plus de 35% en INGMI. Suite à l'introduction d'aides à l'apprentissage, notamment celle décrite dans le paragraphe précédent, les résultats s'améliorent lors de la première session de l'année scolaire suivante : environ 60% en INFO et plus de 50% en INGMI. L'analyse de ces résultats encourage l'introduction de nouveauté pouvant aider les étudiants dans leur apprentissage.

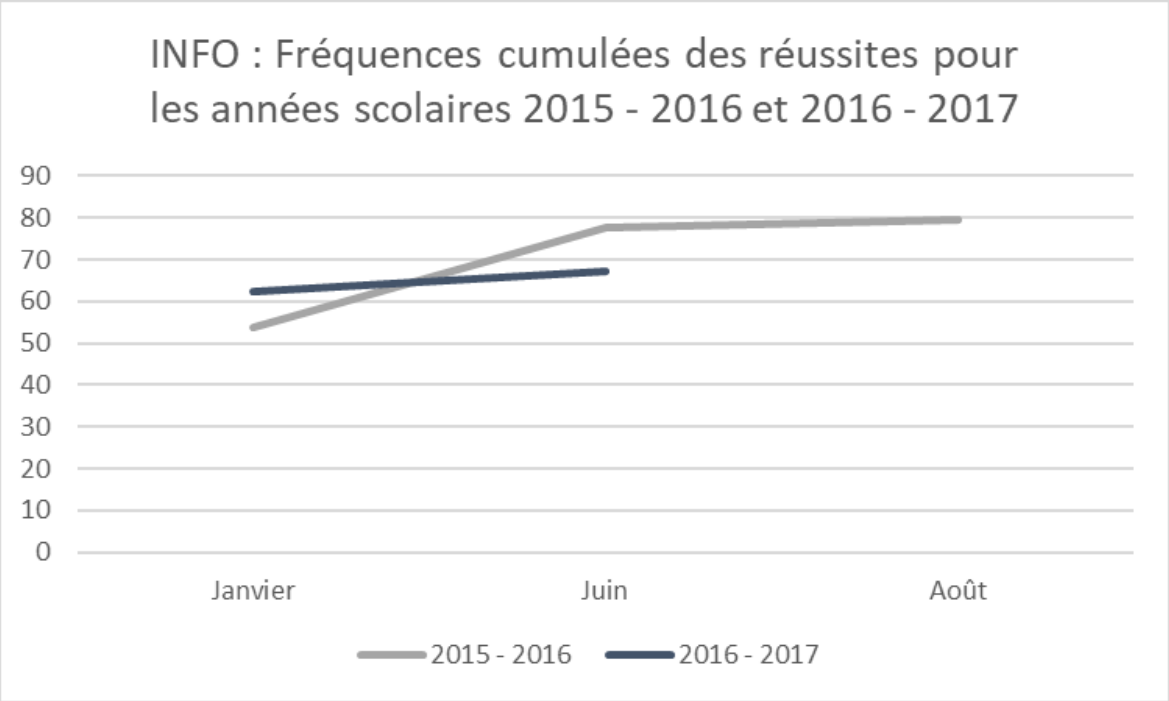


Figure 3: Fréquences cumulées des réussites pour les années scolaires 2015 – 2016 et 2016 – 2017 en fonction des différentes sessions d'examens pour les étudiants INFO.

[7] À l'heure d'écrire ce mémoire, les examens de la session du mois d'août de l'année 2016 – 2017 n'ont pas encore eu lieu.

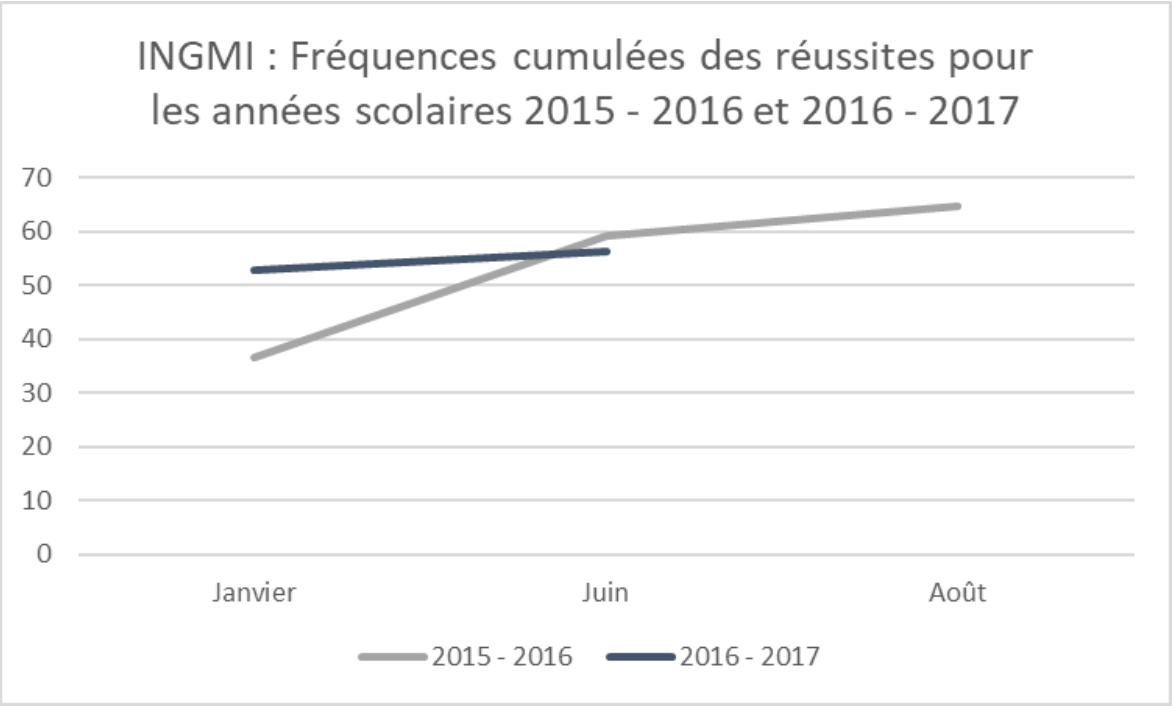


Figure 4 : Fréquences cumulées des réussites pour les années scolaires 2015 – 2016 et 2016 – 2017 en fonction des différentes sessions d'examens pour les étudiants INGMI.

2.2. La notion de Qualité Logicielle

La factorisation n'est pas la seule « preuve » d'une pratique correcte de la programmation. Il faut bien sûr que le code fonctionne. En plus des astuces trouvées par les professeurs afin d'aider les étudiants à mieux appréhender certains concepts, peut-être serait-il utile de leur donner la possibilité de déterminer par eux-mêmes si leurs acquis sont « corrects ». Deux critères principaux permettent de déterminer cette correctitude. Premièrement, il s'agit de s'assurer que le code fonctionne comme attendu, pour cela les étudiants s'essayent à l'utilisation de celui-ci par des tests fonctionnels. Deuxièmement, il est important de valider le contenu du programme, la manière dont le code a été écrit, de tester la présence de bug ou de code inutile, etc. En somme, il s'agit d'évaluer la qualité du programme créé.

Durant la première année de bachelier à l'UNamur (en Sciences Informatiques et en Ingénieur de gestion), la notion de qualité logicielle n'est pas abordée. Celle-ci intervient plus tard, dans le cadre de deux cours organisés en master en informatique uniquement : « Ingénierie du logiciel » (INFOM110) et « Qualité des produits et des processus » (INFOM432).

Le cours INFOM110 est obligatoire et introduit les principes de base du génie logiciel pour « *emmener le développement du logiciel à un niveau d'une vraie ingénierie* »[8], ceci en décrivant les aspects méthodologiques et les outils à disposition afin de garantir la

[8] https://directory.unamur.be/teaching/courses/INFOM110/2017

qualité d'un logiciel (« *fiabilité, facilité d'utilisation, de réutilisation, portabilité, etc.* »[3])

Le second cours est quant à lui optionnel et se concentre sur « *les principes d'évaluation et d'amélioration de la qualité logicielle* ». L'accent est donc mis sur cette notion : description des concepts, des mesures (« *taille, complexité, mesures Orientées Objet, etc.* »[9]), analyses de méthodologies, etc.

Il n'est donc pas attendu des étudiants de première bachelier qu'ils connaissent les notions qui définissent la Qualité Logicielle. Celles-ci pourraient cependant être utiles pour les aider à évaluer leur code de manière globale et à en comprendre les éventuelles faiblesses. Ceci peut se faire de plusieurs manières : l'inspection de code, le test fonctionnel et l'utilisation de mesures, entre autres. Les étudiants réalisant certains de leurs travaux en groupes, il leur est donc possible d'organiser de la relecture de code, du « *peer review* ». Les énoncés distribués durant l'année décrivent les différents aspects fonctionnels attendus, ils n'ont pas d'autre choix que de réaliser des tests pour s'assurer de l'adéquation entre le programme développé et celui décrit dans les énoncés. Par contre, à aucun moment, il ne leur est offert la possibilité de calculer les métriques associées à leur code.

Afin d'être complet dans l'évaluation de la qualité du code produit par les étudiants et donc de leur niveau d'apprentissage de la programmation, il serait dès lors intéressant de mesurer certains aspects non fonctionnels de celui-ci.

[9] https://directory.unamur.be/teaching/courses/INFOM432/2017

Chapitre 3 : Question et Hypothèses de recherche

Dans l'objectif de continuellement améliorer l'enseignement offert aux étudiants, des enseignants tels que M. Frénay cherchent des pistes vers lesquelles se diriger. Ce mémoire tente d'en exploiter une. Démarrant de l'intuition que l'apprentissage de la programmation associée à une sensibilisation à la notion de qualité logicielle permet d'aider les étudiants dans leur phase d'apprentissage, il a été décidé de mettre en place une méthode externe au cours, qui n'interfère que très peu avec celui-ci. Avant de l'y intégrer, c'est-à-dire de l'utiliser de manière active durant le cours, il est nécessaire de la mettre en place et de la valider.

La validation va se faire au travers de la réponse à une question de recherche :

> *L'utilisation d'un outil d'analyse statique de code permet-il à des novices d'améliorer leur apprentissage de la programmation ?*

Celle-ci se décompose en plusieurs hypothèses dont deux vont être traitées dans ce mémoire :

- Une sensibilisation des étudiants à la qualité logicielle (à travers l'utilisation d'un outil d'analyse statique) apporte une plus-value dans le contexte particulier de l'apprentissage de la programmation. (H1)
- Un outil d'analyse statique est utilisable, sans aide extérieure, par des novices. (H2)

La conception de la méthode, quant-à-elle, va se faire de manière itérative et incrémentale, chaque cycle comprenant quatre étapes : mise en place, évaluation, validation et adaptation. Ces cycles seront répétés durant l'année scolaire afin d'obtenir un outil / une méthode qui répondra aux deux hypothèses précitées. Si celles-ci venaient à être confirmées, il serait donc nécessaire de poursuivre le travail dans le futur.

Chapitre 4 : SonarQube comme outil d'analyse statique de code

Ce chapitre a pour objectif de décrire l'outil d'analyse statique qui a été sélectionné dans le cadre de ce mémoire.

Dans un premier temps, il est nécessaire de décrire la démarche mise en place afin d'en choisir un. En effet, une multitude de systèmes de ce type existe sur le marché et il a bien fallu définir des critères afin d'en sélectionner un parmi les autres.

Enfin, une présentation clôturera ce chapitre.

4.1. Choix

Plusieurs systèmes permettant l'analyse statique de logiciels ont été comparés :

- CodeAcy ;
- CodeBeat ;
- SideCI ;
- SonarQube ;
- SourceMeter.

Chacun supporte plusieurs langages de programmation (cfr. Figure 5). Les étudiants-cibles de ce mémoire développant en Python, il était important de s'assurer du support de ce celui-ci. Après comparaison, l'outil qui a été choisi est SonarQube. Les raisons du choix, au nombre de cinq, sont simples.

	Scala	**Php**	**Javascript**	**Python**	**Java**	**Ruby**	**Swift**	**ShellScript**	**C++**	**CofeeScript**	**Css**	**Objective-C**	**Elixir**	**Go**	**Kotlin**	**TypeScript**
CodeAcy	X	X	X	X	X	X	X	X	X	X	X					
CodeBeat			X	X	X	X	X					X	X	X	X	X
SideCI		X	X	X		X	X				X			X		
SonarQube		X	X	X	X				X							
SourceMeter				X	X				X							

Figure 5: Ensemble des langages supportés par les différents outils d'analyse statique de code envisagés.

Premièrement, le logiciel est open-source et propose une version non commerciale qui permet d'analyser du code Python, il convient donc à notre situation.

Deuxièmement, la documentation le concernant est assez exhaustive. Celle-ci est, de plus, soutenue par une communauté importante d'utilisateurs. Cela permet de ne pas rester bloqué face à un problème très longtemps.

Troisièmement, SonarQube est développé en Java et peut donc être déployé sur différentes plateforme. La version d'évaluation est portable (ne nécessite pas d'installation) et emporte une base de données, cela permet de transférer l'environnement d'un ordinateur à un serveur sans problème.

	Gratuit	**Extensible**	**Configurable**	**Open Source**
CodeAcy		X		
CodeBeat			X	
SideCI			X	
SonarQube	X	X	X	X
SourceMeter	X			

Figure 6: Comparaison des différents outils d'analyse statique de code envisagés selon différents critères : la gratuité, l'extensibilité, la configuration et le caractère open-source.

Quatrièmement, il est reconnu dans le monde du développement, est utilisé dans un grand nombre de sociétés et peut être intégré à de multiples environnements de développement.

Cinquièmement, il est configurable et extensible. Il permet d'activer / désactiver la validation de chacune des règles implémentées afin de correspondre au mieux à chaque situation. De plus, si aucune d'entre-elles ne permet d'évaluer une situation particulière, il est possible d'en ajouter de nouvelles via des plugins.

La Figure 6 résume les principaux arguments dans un tableau comparatif avec les différents outils envisagés.

4.2. Présentation

SonarQube est une plateforme de suivi de qualité de code développé par la société suisse SonarSource. Il regroupe un nombre important d'analyseurs statiques de code.

À l'heure d'écrire ces lignes, 18 langages différents[10] sont supportés, pour certains, via une version gratuite, sinon via une version payante.

[10] https://www.sonarsource.com/solutions/deployments/team-grade/

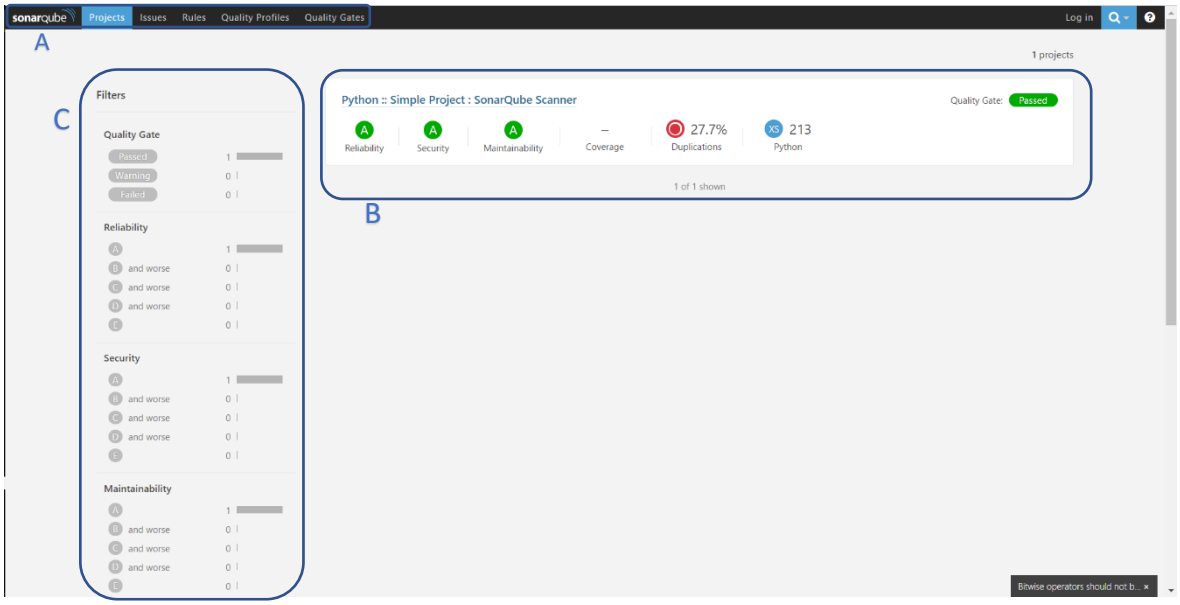


Figure 7: SonarQube - Liste de projets disponibles. Groupe A : menu de l'application. Groupe B : liste de projets analysés par l'instance courante de SonarQube. Groupe C : Filtres disponibles pour trier la liste de projets.

Le support des différents langages se fait au travers de plugins. Pour qu'une instance de SonarQube puisse analyser un langage donné, il est nécessaire de télécharger le plugin au préalable et de le décompresser dans le répertoire adéquat.

Durant les différentes expériences, les étudiants ont accès à certaines pages de la plateforme. La Figure 7 montre le tableau de bord listant l'ensemble des projets qui ont déjà été analysés par SonarQube. Trois parties principales se distinguent. La première, notée A, représente le menu et est présente sur chaque page du portail. La deuxième et la troisième forment la partie dédiée aux projets. Le B se répète autant de fois qu'il y a de projets analysés par SonarQube. Il permet d'obtenir un résumé du résultat de l'analyse pour le projet en question. Le C quant à lui permet d'appliquer quelques filtres sur les caractéristiques affichées dans le résumé (cfr. Figure 7 - Groupe B).

Le tableau de bord d'un projet (Figure 8) reprend l'ensemble des informations disponibles dans le résumé du projet (cfr. Figure 8 - Groupe A). Un complément d'informations (cfr. Figure 8 - Groupe B) est également disponible. Le menu associé au projet (cfr. Figure 8 - Groupe C) permet de naviguer vers les différentes pages liées à celui-ci :

- « *Issues* » liste l'ensemble des remarques que l'analyseur a soulevées ;
- « *Measures* » donne des métriques permettant de qualifier le projet ;
- « *Code* » offre la possibilité de localiser les différentes remarques directement dans le fichier de code ;
- « *Activity* » détaille l'historique des analyses.

La page « *Issues* » permet de garder un point de vue détaillé sur l'ensemble des remarques du projet (cfr. Figure 9). Elles sont regroupées par localisation (fichier de code) et triées par date de création. Il est également possible de sélectionner les remarques à afficher en appliquant des filtres.

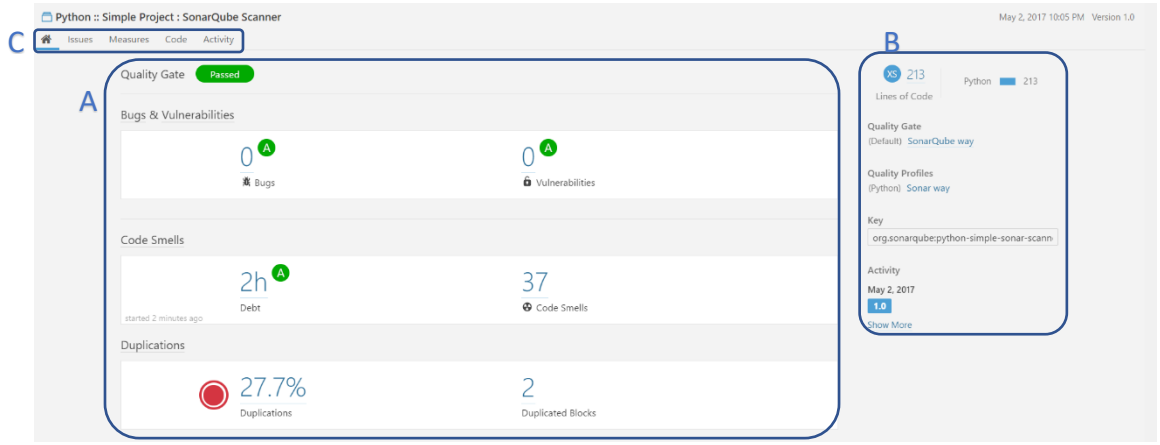


Figure 8: SonarQube - Tableau de bord d'un projet. Groupe A : résumé du résultat de la dernière analyse réalisée. Groupe B : résumé de métadonnée liées au projet. Groupe C : Menu permettant de naviguer dans les différentes pages du projet disponibles.

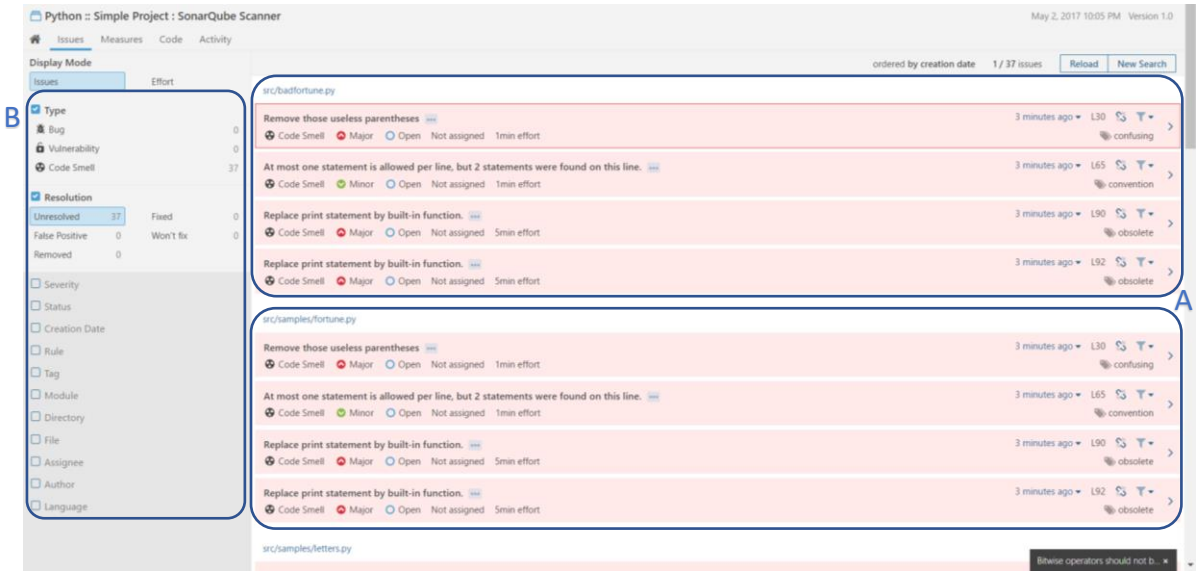


Figure 9: SonarQube - Liste de remarques représentant les violations de règles rencontrés durant l'analyse du code du projet sélectionné (en haut à gauche de l'image, ici : « Python :: Simple Project : SonarQube Scanner »). Groupe 1 : les remarques sont regroupées par fichier. Groupe B : filtres disponibles pour sélectionner les remarques à afficher en A.

Une analyse statique calcule un ensemble de mesures permettant d'évaluer de manière formelle la qualité du code analysé. La Figure 8 montre un résumé de celles-ci. Chacune de ces mesures est décrite par des détails accessibles en un clic. La page « Measures » offre plus de détails afin d'évaluer la qualité du travail fourni (cfr. Figure 10).

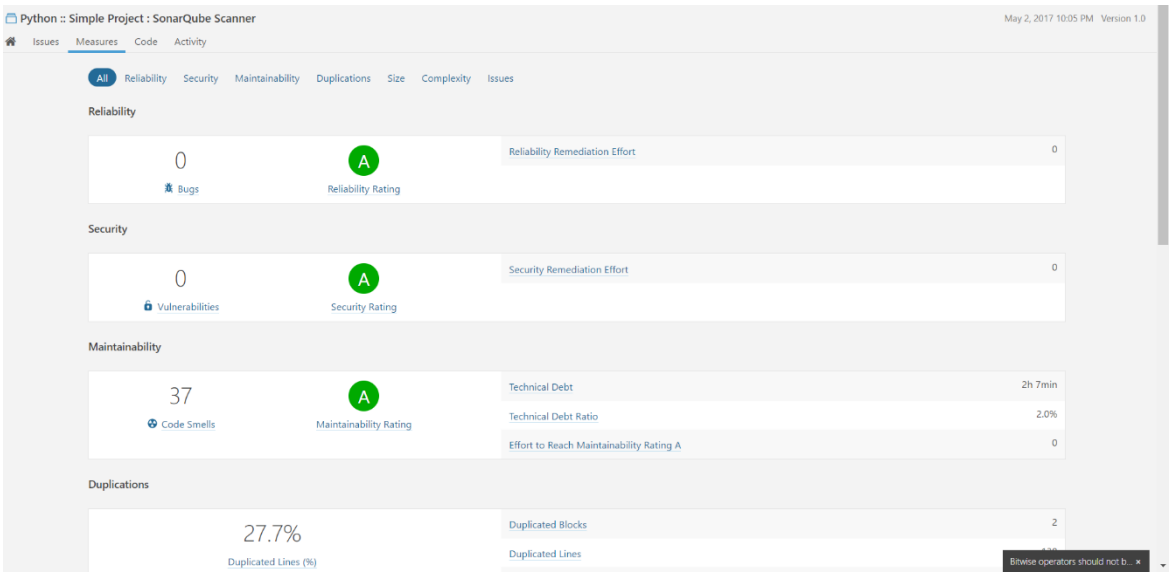


Figure 10: SonarQube - Quelques mesures obtenues suite à l'analyse du code du projet en cours.

Afin de contextualiser les remarques, la vue « *code* » (Figure 11) permet de les replacer visuellement dans le fichier concerné. Celle-ci se compose de l'ensemble des lignes de code auxquelles sont ajoutées les remarques générées par l'analyse. Le détail de la remarque est inséré après la ligne référencé et est semblable à ceux qui composent la liste représentée par la Figure 9. Le « groupe A » représente un ensemble ligne-remarque. Le code concerné par la remarque est souligné par un trait rouge. Il est possible de sélectionner la remarque afin de faire ressortir des informations la concernant. Par exemple, la ligne concernée est surlignée par une couleur rose.

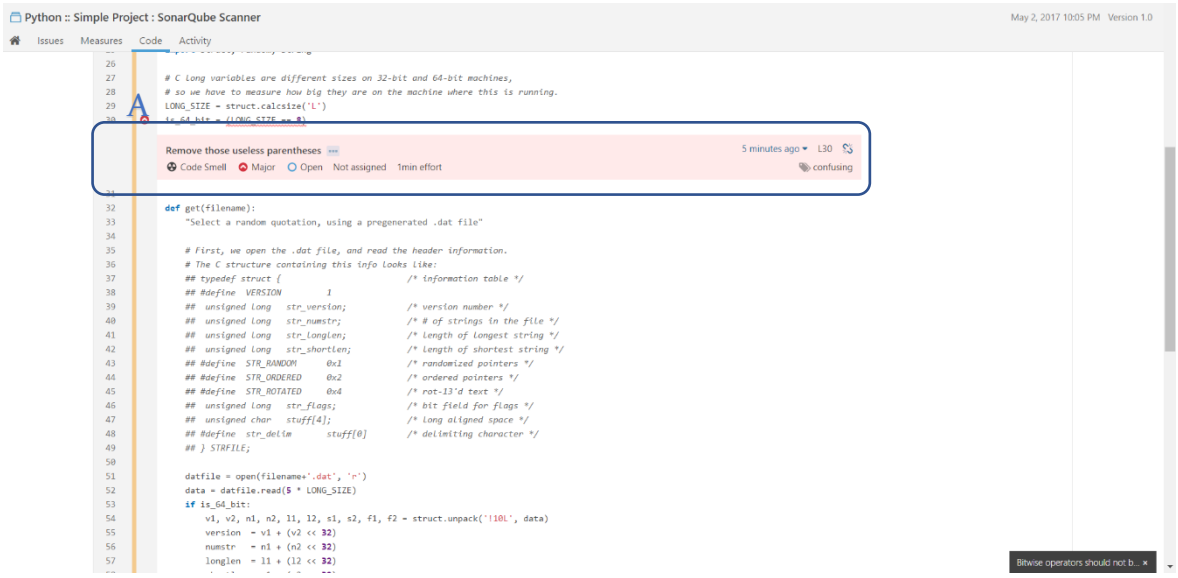


Figure 11: SonarQube - Code inspecté durant l'analyse. Groupe A : décoration du code de la ligne 39 par le détail de la remarque générée suite à la violation de la règle « Remove those useless parentheses ».

Un historique des analyses effectuées peut être consulté par la vue « *Activity* » (Figure 12). Il peut parfois être utile de prendre connaissance des informations de la dernière analyse : la date et l'heure de la dernière analyse, son statut.

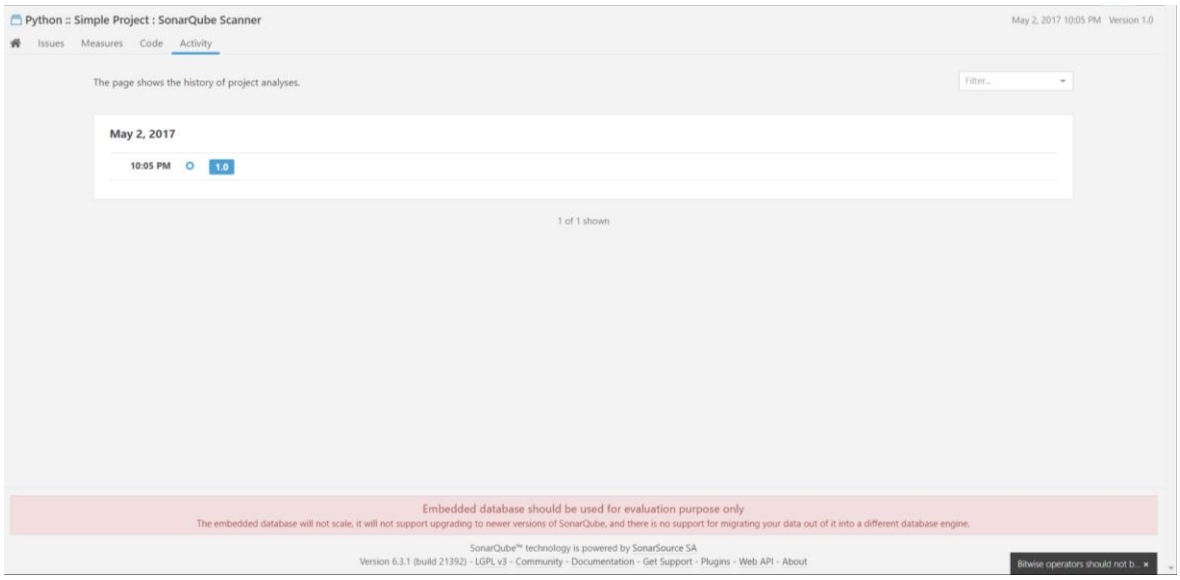


Figure 12: SonarQube - Historique des analyses réalisées pour le projet sélectionné.

Chapitre 5 : Méthodologie de recherche

Deux expériences sont nécessaires pour vérifier les hypothèses précédemment définies et ainsi apporter des éléments de réponse à la question de recherche inspirant ce mémoire.

La première expérience (notée ici EXP1) consiste à mettre les étudiants en contact avec l'outil d'analyse statique et à observer leur comportement face à celui-ci.

Cette expérience se déroule dans le cadre d'un mini-projet du cours INFOB131 (cfr. Chapitre 2 : Contexte et problématique) et en présence d'une assistante de ce cours. Les étudiants, par groupes de quatre à six, sont invités à venir découvrir durant 20 minutes le résultat de l'analyse préalable de leur code par l'outil[11]. Le mini-projet n'étant pas terminé, la version de code utilisée est intermédiaire. Au total, 19 groupes participent à l'expérience pour un total de 81 étudiants.

Les données récoltées sont multiples et complémentaires. Ainsi, chaque passage de groupe est filmé[12] dans son entièreté. Toute observation effectuée est annotée (objet de l'observation et minutage), de façon à faire correspondre celle-ci à un moment précis de l'enregistrement vidéo. Après un moment de totale autonomie pour les étudiants, des questions précises sont posées faisant référence à des caractéristiques (fonctionnalités, interfaces, etc.) de l'outil. Aux termes des 20 minutes, chaque étudiant est invité à remplir individuellement un questionnaire visant à mesurer sa satisfaction et son implication dans l'expérience. Enfin, les analyses du code fourni par chacun des groupes pour l'expérience et celles du code remis par les groupes pour évaluation du mini-projet (code final) restent également exploitables.

En ce qui concerne le déroulement de l'expérience EXP1, celui-ci suit un schéma précis, identique pour l'ensemble des groupes. Une fois installés dans le local prévu à cet effet, les étudiants se voient expliciter ce qu'il est attendu d'eux durant les 20 prochaines minutes. Il s'agit ici de leur permettre de rentrer rapidement dans le « jeu ». Ainsi, les informations suivantes leur sont fournies :

- Une présentation rapide de l'outil d'analyse statique et de ses principales interfaces ;
- Le résultat de l'analyse préalable, par l'outil, de leur code ;
- Une brève distinction entre test fonctionnel et analyse statique.

Les étudiants sont invités à verbaliser au maximum leurs interactions avec l'outil : énoncer les remarques qui font l'objet de leur discussion, expliciter ce qu'ils en

[11] Soumission effectuée par l'auteur.

[12] Les étudiants ont tous marqués leur accord via un formulaire signé au préalable.

comprennent, décrire où ils en sont dans le fichier, préciser sur quel lien ils cliquent, etc.

Si les interactions avec les observateurs (l'auteur et l'assistante) sont réduites au minimum (remarques à approfondir, relances, comportements étonnants, etc.), une assistance leur est toutefois accordée en cas de problèmes de compréhension ou de traduction, par exemples, entravant le bon déroulement de l'expérience.

Une période de questions-réponses est prévue en fin de rencontre. Il est notamment demandé aux étudiants leur interprétation de certaines informations détaillant chaque remarque (cfr. Figure 13).

- Description : courte description de la règle qui n'a pas été respectée ;
- Catégorie : différentes catégories disponibles : « *Bug* », « *Vulnerability* », « *Code Smell* » ;
- Gravité : différentes gravités disponibles : « *Blocker* », « *Critical* », « *Major* », « *Minor* », « *Info* » ;
- Dette technique générée : durée de la dette générée suite au non-respect de la règle ;
- Date apparition : date de l'apparition de la remarque dans le résultat de l'analyse ;
- Numéro de ligne
- …

Figure 13 : SonarQube - Détails d'une remarque générée suite à la violation de la règle « Refactor this code to not nest more than 4 « if », « for », « while », « try » and « with » statements. » en ligne 914 (voir texte pour détail).

Par exemple, il peut être demandé ce que représente la durée indiquée (i.e. : 10 min effort), ou encore ce qu'ils comprennent par « *Major* » et s'ils sont d'accord avec cette classification. Pour aller plus loin, il leur est demandé dans quel ordre ils traiteraient les remarques, sur base de quels critères, s'ils accorderaient plus d'importance à l'un ou l'autre critère et pourquoi.

Le questionnaire individuel est composé de quelques questions permettant aux étudiants de donner leurs impressions :

1) *Connaissez-vous l'outil qui a été utilisé ? (oui/non) Si oui, quel est son nom ?*
2) *Selon vous, cette expérience était (…) ?*
   Il est proposé aux étudiants de répondre via une échelle de Likert composée des valeurs : très utile – utile – inutile (dans les conditions proposées) – vraiment inutile. Il est également demandé de justifier le choix.
3) *Avez-vous compris les remarques faites par l'outil ?*
   Il est proposé aux étudiants de répondre via une échelle de Likert composée

des valeurs : oui, toutes – oui, partiellement – non, pas vraiment – pas du tout
Il est également demandé de justifier le choix et / ou de donner un exemple.

4) Trois remarques à décrire :
Access to member before its definition ; *Functions should not be too complex* ; *Undefined variable.*

La deuxième expérience (notée EXP2) consiste à mettre à disposition des étudiants, durant six semaines, l'outil d'analyse statique (à travers un environnement personnalisé – cfr. Chapitre 7 : Préparation de la seconde expérience), sans obligation de l'utiliser. Cette seconde expérience prend place dans le cadre du cours INFOB132, durant le projet de développement. Les étudiants travaillent alors par groupe de trois. Au total, 30 groupes sont susceptibles d'utiliser l'outil.

Chapitre 6 : Première expérience – Une utilisation « observée » de l'outil

La première expérience a eu lieu au mois de novembre. Elle a eu pour objectifs de : « *mettre les étudiants en contact avec l'outil d'analyse statique et à observer leur comportement face à celui-ci* » (cfr. Chapitre 5 : Méthodologie de recherche).

Ce chapitre consiste en l'analyse des données récoltées durant cette expérience : les réponses aux questionnaires d'après séance, les résultats des analyses de code des étudiants à deux périodes et les enregistrements vidéo de l'expérience supportés par les annotations en direct des observateurs.

En fin de chapitre, les conclusions intermédiaires permettront de donner des indications pour le travail à réaliser dans la suite du mémoire.

6.1. Données collectées : analyse et discussions

L'expérience à produit plusieurs types de résultats :

- Les questionnaires remplis ;
- Les résultats des analyses de code des étudiants avant l'expérience et en fin de mini-projet ;
- Les enregistrements vidéo agrémentés des annotations en direct.

6.1.1. Analyse des réponses aux questionnaires

D'une manière générale, SonarQube est inconnu des étudiants. Excepté deux étudiants, ils ont tous répondu non à la première question. Dans les deux étudiants connaissant l'outil, un l'a nommé SonarSource et l'autre ne l'a pas nommé. Parmi les étudiants ne connaissant pas l'outil, un étudiant a indiqué avoir déjà utilisé un outil du même genre.

On remarque, grâce aux réponses aux questionnaires, que l'expérience a semblée utile pour presque tous les étudiants : 37 l'ont trouvé très utile et 44 utile. Il reste 2 étudiants ayant répondu inutile et vraiment inutile.

Pour ce qui est de la compréhension des remarques faites par SonarQube, 37 étudiants déclarent les avoir toutes comprises, 46 estiment avoir partiellement compris les remarques. Un seul étudiant a, pour sa part, indiqué « *pas du tout* ».

6.1.1.1. Utilité de l'expérience

Les étudiants ayant trouvé l'expérience très utile se justifient en déclarant que SonarQube peut permettre de donner un résumé de la situation : « *Ça permet de nous rendre compte de notre avancement dans l'élaboration de notre code et d'avoir un résumé clair …* », ou leur fait gagner du temps : « *Permet de gagner beaucoup de temps qu'on aurait passé à relire le code* ». SonarQube permet également d'apporter un regard externe : « *… il y a des erreurs qu'on n'aurait pas pu détecter, …* », de pointer de mauvaises habitudes : « *Elle a permis de faire ressortir des mauvaises habitudes que j'ai*

pris en codant et me faire comprendre comment les corriger » ou encore d'aider à simplifier le code : « *Cela nous a permis de voir où on peut simplifier notre code* ».

En ce qui concerne les étudiants ayant trouvé l'expérience utile, SonarQube peut aider à comprendre des erreurs : « *Cela nous a permis de comprendre nos erreurs plus vite que si on avait exécuté avec python* » ou à rendre leur code plus propre : « *Les erreurs que le programme a diagnostiquées n'étaient pour nous pas de grosses erreurs mais seulement pour avoir un code plus propre* ». Certains ciblent la factorisation : « *L'expérience nous a permis de voir dans le code les endroits où on pouvait le factoriser* ». La notion de convention de nommage intervient également : « *Permet de mettre en évidence des erreurs qui pourraient poser problème pour travailler dans le code à plusieurs par après (Majuscule, minuscule pour les variables)* ».

L'étudiant ayant répondu inutile se justifie par « *Dans le cadre de ce mini-projet, les problèmes de complexité du programme ne sont pas vraiment importants* ».

L'étudiant ayant répondu vraiment inutile se justifie de façon étonnante par : « *Nous avions du mal à faire ce mini-projet donc le fait qu'on nous montre nos erreurs aide beaucoup* ».

Un étudiant en particulier s'inquiète de voir les étudiants devenir paresseux : « *… mais le programme peut rendre les étudiants paresseux. C'est-à-dire qu'ils ne feraient plus d'effort parce que de toute façon il y a un programme pour tout revérifier* ». Cette remarque montre une certaine confiance en l'outil.

6.1.1.2. Compréhension des remarques

Pour les étudiants ayant partiellement compris les remarques, certains déclarent avoir eu besoin d'explications supplémentaires : « *Toutes sauf la dernière que je n'aurai pas pu comprendre sans explication* ». Certaines remarques n'ont pu être comprises car la notion abordée n'a pas encore été vue au cours : « *Les remarques sur la complexité, mais nous n'avions pas vu ça en cours* ». Certains proposent même des évolutions : « *Je trouve que si on parle d'un 'if' ou d'un 'or' etc. il devrait être mis dans une autre couleur pour éviter des confusions* ». Pour certains étudiants, les informations proposées par SonarQube ne sont pas intuitives : « *Le temps affiché et le type d'erreur ne sont pas vraiment intuitifs* ». Certains groupes ont reçu une erreur contenant une expression régulière et globalement celle-ci n'a pas été comprise : « *Certaine partie comme [a-z_][a-z0-9_] n'est pas vraiment compréhensible* », « *… [a-z_][a-z0-9_] compris uniquement après explication* ».

Plusieurs éléments reviennent indépendamment de la réponse : l'anglais, l'indication de la ligne cible et la contextualisation de la remarque. En ce qui concerne l'anglais, certains ont des difficultés : « *…. Je pense que c'était plutôt un problème avec l'anglais* », « *Il y a des erreurs qu'on n'a pas compris à cause de l'anglais …* », « *J'ai dû avoir des explications car certains mots en anglais étaient compliqués, mais après traduction, j'ai compris* », « *Nous ne connaissions pas la signification du mot « Merge »* » ; d'autres

ont des facilités : « *Les remarques sont accessibles avec un anglais assez simple donc utilisable par un grand nombre de gens* ». L'indication de la ligne cible : « *Le fait que les lignes soient indiquées est positif* », la contextualisation de la remarque : « *Les remarques du programme étaient claires, possibilité de les voir dans le code avec les éléments qui posent problème surlignés en rouge* ».

L'étudiant ayant indiqué « pas du tout » se justifie de manière étonnante par : « *Le logiciel est clair et bien utile, très simple d'utilisation et met facilement en évidence les erreurs* ».

6.1.2. Analyse des listes de remarques générées par SonarQube pour code des étudiants

Lorsque SonarQube termine l'analyse du code d'un groupe d'étudiants, il produit une liste de remarques critiquant la qualité du code. En réalisant l'analyse avant et après l'expérience, il devient possible de comparer les résultats produits. Une seconde analyse a donc été réalisée avec le code que les étudiants ont remis pour l'évaluation finale de leur mini-projet.

Il n'est pas évident de réaliser une comparaison qualitative du projet fourni pour l'expérience et de sa version finale. Il est par contre plus facile de comparer des nombres. Dans cet objectif, une matrice a été réalisée en agrégeant le nombre de remarques similaire par groupe (cfr. Figure 14).

Dans cette matrice, chaque ligne représente une remarque différente. Les colonnes, quant à elles représentent les groupes. Chaque colonne a été partagée en deux parties : la partie de gauche représente l'évaluation réalisée pour l'expérience, ci-après « phase 1 » et la partie de droite représente l'évaluation réalisée avec le code final, ci-après « phase 2 ». Le nombre indiqué dans une cellule de la matrice représente le nombre d'occurrences de la remarque dans le résultat de l'analyse.

Les cellules de la phase 2 sont colorées en fonction de l'évolution du nombre d'occurrences de remarques identiques dans l'analyse qui s'y rapporte par rapport à l'analyse en phase 1 : le vert représente une diminution du nombre d'occurrences ; le rouge signifie au contraire que l'analyse en phase 2 contient plus d'occurrences que la première. Enfin, les dernières lignes permettent de comparer les évolutions entre le nombre total de remarques par groupe et le nombre de remarques différentes par groupe pour chaque étape.

```
1    def move_pawn(player, position_1, position_2) :
2        """Move the pawn of a player
3        Parameters
4        ----------
5        player : ID of the player (1 or 2) (str)
6        position_1 : initial position of the pawn (tuple)
7        position_2 : final position of the pawn (tuple)
8        Note
```

```
9       ----
10      There must be pawns on the board
11      The pawn can not move diagonally
12      The box must be availible
13      """
14
15      Matrix = load_game("matrix")
16      Player = load_game("dictionary")
17      #check if the pawns exists
18      if Player[player]["number_pawns"] ==0 :
19          x1 = position_1 [0]
20          y1 = position_1 [1]
21          x2 = position_2 [0]
22          y2 = position_2 [1]
23          #check if they are a pawns to the first coordinates and nothing to the second
24          if Matrix[x1][y1]==player and Matrix[x2][y2]=='0' :
25              #check if the pawns moves only a box at the same time
26              #check if the pawn doesn't move diagonally
27              if ((x2==x1+1 or x2==x1-1) and (y2==y1)) or ((y2==y1+1 or y2==y1-1) and (x2==x1)) :
28                  Matrix[x1][y1]='0'
29                  Matrix[x2][y2]=player
30                  save(Matrix, Player)
31                  sandwich(player, position_2)
32              else :
33                  print "you cannot move more than one box a time"
34          elif Matrix[x1][y1]!=player :
35              print "you don't have any pawn on this box"
36          else :
37              print "this box isn't available"
38      else :
39          print "you don't have put all your pawns on the board"
40      Player = load_game("dictionary")
41      if Player['2']["pawns_on_board"] <=1:
42              print "player 1 has win the game, congratulation!!"
43      elif Player['1']["pawns_on_board"] <=1:
44          print " player 2 has win the game, congratulation!!"
```

Code 2: Groupe 2 - Exemple de fonction mélangeant des variables avec différentes conventions de nommage.

L'observation de ces lignes de résumé permet de se rendre compte de l'évolution globale réalisée par une majeure partie des groupes. En effet, sur les 19 groupes participant à l'expérience, seulement quatre ont régressé. De plus, le code utilisé pour la deuxième étape est plus conséquent que celui utilisé lors de l'expérience. En effet, la deuxième étape correspond au code final du mini-projet des étudiants, donc une certaine indulgence peut être accordée aux groupes ayant régressé.

```
1   if ii+2 <= size :
2       if game ['playboard'][str(ii+1) + ','+ j] == 'x' and game ['playboard'][str(ii+2) + ',' + j]== 'o':
3           if ii+1 != (size+1)/2 or jj != (size+1)/2 :
4               game ['playboard'][str(ii+1) + ',' + j ]= '.'
5               game['player_2'] ['nb_pawns'] -= 1
```

```
6           print 'you succeded to take a pawn of the other player'
7           game ['Counter'] =0
```

Code 3: Groupe 16 - Exemple de code contenant plusieurs "if" imbriqués.

Certains nombres présents dans cette matrice peuvent paraître surprenant : 12 occurrences de la remarque « *Rename this local variable "xxx" to match the regular expression ^[_a-z][a-z0-9_]*$.* » pour le groupe 2, 16 occurrences de la remarque « *Merge this if statement with the enclosing one* » pour le groupe 16 et 41 occurrences de la remarque « *Remove those useless parentheses* » pour le groupe 19. Et pour cause, ils témoignent de vraies mauvaises habitudes que les étudiants ont, déjà, pris (cfr. Code 2, Code 3 et Code 4).

```
1   pawns = { ('pawnstoplaceP1'): (table**2-1)/2,  ('pawnstoplaceP2'): (table**2-1)/2,
2                   ('turns'): 0,  ('hitP1'): 0, ('hitP2'): 0, ('totalpawnsP1'): (table**2-1)/2,
3                   ('totalpawnsP2'): (table**2-1)/2 }
```

Code 4: Groupe 19 - Exemple de code contenant des parenthèses inutiles.

À l'opposé, certains résultats sont particulièrement réjouissant. En effet, trois groupes ont réussi à produire du code ne violant aucune règle et quatre autres groupes ont réussi à n'en violer qu'une seule.

Il est important de signaler que les étudiants n'ont pas eu d'autres accès aux résultats d'analyse de leur code que durant l'expérience EXP1.

	Groupe 1		Groupe 2		Groupe 3		Groupe 4		Groupe 5		Groupe 6		Groupe 7		Groupe 8		Groupe 9		Groupe 10		Groupe 11		Groupe 12		Groupe 13		Groupe 15		Groupe 16		Groupe 17		Groupe 18		Groupe 19		Groupe 20	
Merge this if statement with the enclosing one		1	4						2				4	3	4		8	4			2		4		4				1	16			4				1	
Refactor this code to not nest more than 4 "if", "for", "while", "try" and "with" statements		1							1				1	4			4	4	1	1			1				1			4							2	1
Rename this local variable "**xxx**" to match the regular expression ^[_a-z][a-z0-9_]*$.			10	12			8						3																				1					
Function has a complexity of **000** which is greater than 20 authorized.			1	1	1	1	1						3	2			1	1	1	1	2	2							1		2							
Remove those useless parentheses					2	2	1	3			2	1									1		1	1					1		1	8	4	6	41	2		
File "**xxx**.py" has **000** lines, which is greater than 1.000 authorized. Split it into smaller files.								1																								1						
Either merge this branch with the identical one on line "**000**" or change one of the implementations.																			3					3			1					3						
Remove the code after this "return".																					1					1	1		1	1								
Function "**xxx**" has **000** parameters, which is greater than the 7 authorized.																									1		1					1			1			
Remove this commented out code.																													1									
At most one statement is allowed per line, but **000** statements were found on this line.																															1			1				
Rename this parameter "**xxx**" to match the regular expression ^[_a-z][a-z0-9_]*$.																																4						
Nombre total de remarques	0	2	15	13	3	3	10	4	3	0	2	1	11	9	4	0	13	9	5	2	6	2	6	4	5	1	4	0	3	23	1	3	21	6	6	42	5	1
Nombre de remarques différentes	0	2	3	2	2	2	3	2	2	0	1	1	4	3	1	0	3	3	3	2	4	1	3	2	2	1	4	0	3	5	1	2	6	3	1	2	3	1

Figure 14: Tableau regroupant l'ensemble des règles violées durant toutes les analyses réalisées pour l'expérience EXP1 en première colonne. Pour chaque groupe, deux colonnes permettent de dénombrer le nombre d'occurrences de chaque remarque avant l'expérience (première colonne) et lorsque le code a été validée par les étudiants (deuxième colonne). Les deux dernières lignes résument la situation.

6.1.3. Analyse des enregistrements vidéo

6.1.3.1. Méthodologie d'analyse

Les séquences durant lesquelles les étudiants ont eu accès au résultat de l'analyse de leur code par SonarQube ont été filmées. Il a donc été possible de les retranscrire. Ceci a créé des dialogues qui ont ensuite pu être analysés. L'analyse a suivi le schéma proposé par Christophe Lejeune dans son ouvrage « Manuel d'analyse qualitative » (Lejeune, 2014).

D'un point de vue pratique, les retranscriptions ont été effectuées à l'aide d'un ordinateur, les vidéos ont été visionnées via VideoLan[13] et le texte a été retranscrit manuellement avec Microsoft Word[14]. Une fois terminée, chaque retranscription a été imprimée sur du papier blanc avec une marge gauche plus importante que d'habitude.

Dans un premier temps, pour le premier texte à analyser, il faut prendre le temps de plonger, de s'immerger, dedans. Pour ce faire, l'auteur du livre propose d'annoter chaque ligne du texte par un mot, sans tenir compte de la notion de phrase. Cette annotation se doit d'être descriptive et exhaustive ; il n'est pas nécessaire de chercher « quelque chose d'intéressant ». Celles-ci ne seront d'ailleurs pas utilisées. Le côté pratique pousse à utiliser la marge de droite pour y indiquer le mot descriptif (cfr. Figure 15).

L'étape suivante est à reproduire pour chaque dialogue retranscrit et consiste à étiqueter des passages du texte. Pour cela, au contraire de la première étape, il faut partir à la recherche d'éléments concrets, qui ont de la valeur. Cela peut être un mot, une expression, une phrase ou même un dialogue. L'important est qu'il faut retranscrire un concept, idéalement par une courte déclaration (un à deux mots). Cette partie permet de créer le contenu, la base qui sera utilisée pour comparer les différents dialogues. La marge gauche augmentée permet d'avoir une aisance dans l'écriture, et surtout la réécriture (cfr. Figure 15).

Afin de clôturer l'analyse d'un dialogue, il est nécessaire de regrouper les étiquettes par thème. Ceci permet de faire émerger des catégories, qui seront utilisées lors de la discussion des résultats. Pratiquement, toutes les étiquettes ont été retranscrites manuellement sur une feuille A4. Ensuite, il reste à essayer de les rassembler jusqu'à obtenir un résultat cohérent. Soit les catégories déjà existantes peuvent être réutilisées, soit il est nécessaire d'en créer de nouvelles.

Il n'a pas été nécessaire d'analyser l'ensemble des dialogues de tous les groupes. En effet, après quelques-uns, on se retrouve dans une situation de saturation. C'est-à-

---

[13] https://www.videolan.org/vlc/index.fr.html

[14] https://products.office.com/fr-be/word

dire que les catégories déjà trouvées dans les analyses précédentes suffisent à classer les étiquettes définies, qu'il n'est pas nécessaire d'en créer de nouvelles.

découverte des erreurs

Ét1 : C'est quand même pratique, on avait vu des erreurs qu'on avait pas fait attention en relisant le code, du coup...

Comprendre les remarques

Ét4 : C'est facile à comprendre aussi

Ét5 : Oui, ça c'est vrai

Focaliser sur le nombre complexité

Ét3 : Moi je pense que simplement avec le fait que la dernière dise que « Function has a complexity of 22 », le problème c'est que on ne sait pas vraiment ça veut dire quoi 22 supérieur à 20

Comprendre est suffisant

VB : Et est-ce que tu estimes que c'est important ou pas ? Est-ce qu'il faudrait absolument savoir ce que représente le 22 ou pas ?

Ét3 : Peut-être pas, parce que là, avec le mot « Complexité » on sait déjà à peu près ce que ça veut dire

VB : D'accord

Curiosité

Ét3 : Mais après, peut-être les chiffres savoir cd que c'est est un peu plus ...

VB : OK

Ét3 : plus difficile

VB : Moi je n'ai pas d'autres questions donc on peut s'arrêter là

JH : Attends, moi j'ai une question par rapport au fait que ça, je ne sais pas s'il vous l'a dit, mais le code ne s'exécute pas ici ? Je ne sais pas si votre code tourne ou pas

Ét5 : Oui, il fonctionne

complémentaire Sympa Canopy

JH : Est-ce que vous trouvez que c'est complémentaire aux erreurs que Canopy vous signale quand vous faites des erreurs d'écriture ?

Amélioration ciblées

Ét5 : Clairement oui, parce que les trois premières où ça met que... Enfin on a trois « if » qui retournent à chaque fois un « false », enfin je pense qu'on était tous tellement dans le code et un peu trop plongé dedans qu'on n'a même pas vu que c'était totalement inutile donc c'est vrai que pour des trucs comme ça c'est vraiment hyper intéressant pour pouvoir factoriser le code

JH : Et ici le code vous signale cinq erreurs, est-ce que pour autant vous les prenez toutes comme étant... Enfin, vous faites une confiance aveugle en ce système ou il y a des erreurs, un peu comme les chiffres là, un peu moins... Pas moins d'accord, mais où vous vous dites « C'est mitigé là » ? Parce qu'il met « Majeur » sur toutes les erreurs par exemple ici ; est-ce que vous trouvez que toutes ont, effectivement, une importance « Majeure » ?

Améliorations triviales

Ét5 : Ben je pense que oui parce que, enfin, pour les trois premières et la dernière clairement parce qu'on a vu qu'on pouvait énormément factoriser en ne changeant rien du tout au code donc c'est clair qu'il y a un problème dans le code à ce niveau-là et que c'est pas compliqué a changer et que ça peut à mon avis, faire tourner le code plus facilement. Maintenant pour la quatrième, je n'en ai encore aucune idée parce que je ne sais pas encore comment on va pouvoir changer ça

Utiliser pour factoriser

Ét3 : Après moi je pense que, franchement, ça aide... enfin moi comme il dit à factoriser un peu le code et à ne pas répéter plusieurs fois la même chose. Mais comme il dit aussi après, il faut aussi voir comment est-ce qu'on peut utiliser ça pour améliorer le code

utiliser pour améliorer le code

Ét1 : Mais c'est pratique parce que quand on a le nez dans le code, on fait pas vraiment attention, on sait quand même avoir un regard externe sur notre code et de savoir, d'avoir une nouvelle vision en fait dessus.

Solutions pour améliorations triviales

certaines remarques permettent de trouver des solutions triviales

↳ indiquent précisément l'amélioration

3/5

Figure 15: Exemple de retranscription (contenu de la page), d'annotation (marge de droite) et d'étiquetage (marge de gauche). Technique utilisée pour l'analyse des dialogues filmés durant l'expérience EXP1.

6.1.3.2. Résultats de l'analyse des séquences vidéo

Comme indiqué précédemment, il n'a pas été nécessaire d'analyser l'ensemble des groupes. La saturation recherchée a été obtenue après quatre groupes. Cette rapidité peut s'expliquer par le fait que le déroulement a été similaire pour tous les groupes. De plus, le contexte dans lequel les étudiants évoluent est identique : ils suivent les mêmes cours à l'Unamur et se retrouvent donc avec la même base théorique, ils réalisent le même mini-projet, etc. Enfin, les questions posées durant l'expérience ont évolué mais sont restées sensiblement semblables.

Ces analyses ont permis d'identifier trois axes de discussion différents :

1) La correction des remarques générées par SonarQube ;
2) L'interface de SonarQube ;
3) Les utilités de SonarQube.

6.1.3.2.1. Correction des remarques

Le schéma mental que les étudiants ont reproduit concernant la correction d'une remarque proposée par SonarQube passe par trois étapes. Dans un premier temps, les étudiants la lisent et essayent de la comprendre. Les remarques étant écrites en anglais, il est nécessaire pour eux de les traduire avant toute autre chose.

De manière générale, les étudiants en groupe sont en mesure de traduire correctement les remarques. Il arrive que certains passent beaucoup de temps sur les détails et aient besoin d'être recadrés, redirigés, mais cela ne nuit pas à la compréhension.

Certains étudiants ne se sentent pas dérangés par l'anglais, mais pour beaucoup d'entre-eux, cela pourrait devenir problématique. Des termes spécifiques tels que « Merge » ont posé des difficultés. Pour certains cas particuliers, la simple traduction se révèle même incompréhensible : « Merge this if statement with the enclosing one » devient « Fusionne cette phrase si avec celle qu'elle contient » ; au moins deux problèmes sont rencontrés ici, « statement » n'est pas interprété comme « instruction », donc le « if » n'a pas l'importance désiré et se fond dans la phrase. La contextualisation permet néanmoins à certains groupes de trouver le sens de la phrase lorsque la traduction fait défaut.

Une fois la traduction terminée, place à l'interprétation ; il est nécessaire de pouvoir contextualiser la remarque avant de commencer la dernière étape, la recherche de correction.

Durant cette phase, deux types de remarques ont émergés : les remarques dites « évidentes » et les « complexes ». Les remarques évidentes donnent la solution dans la description, par exemple « Merge this if statement with the enclosing one », « "<>" should not be used to test inequality », etc. À l'opposé, pour pouvoir trouver le point ciblé par une remarque dite « complexe », il est nécessaire de la comprendre et de

pouvoir la contextualiser. Cette contextualisation n'est pas toujours chose aisée, les étudiants ne remarquent que très peu les aides visuelles de SonarQube. De plus, il arrive qu'ils lisent trop vite ou partiellement la description, ce qui les empêchent de comprendre correctement et les oblige à être aidés, parfois même plusieurs fois. Cependant, même s'ils n'utilisent pas l'ensemble des aides que propose SonarQube, leur attention est attirée sur une zone du code contenant vraisemblablement une faiblesse : celle où se situe la remarque. Pour certaines situations relativement simples, cette analyse par l'observation peut suffire.

La compréhension d'expressions régulières ou de notions telles que la complexité s'avère surprenante pour des étudiants de leur niveau. Bien entendu, tous les groupes n'ont pas généré ce genre de remarques et tous ceux les ayant générés ne les ont pas compris. Certains interprétaient la complexité par la taille du nom de la fonction, ou par le nombre de paramètres.

D'une manière générale, les étudiants ont essayés de trouver une solution aux remarques proposées par SonarQube. Pour certaines de celles-ci, ils se sont étonnés car la notion abordée n'a pas été vue au cours (complexité, profondeur de code, …). Pour le reste, les solutions des remarques « évidentes » sont trouvées presque directement après avoir compris la description. Les étudiants sont même motivés de trouver la solution à des remarques qui pourraient améliorer les performances de leur code. Certains groupes ont néanmoins eu besoin d'aide pour chaque étape de la réflexion. Des notions de base étant même parfois confondues : assignation et fonction, simplification et factorisation, …

Lorsque la correction d'une remarque demande trop de réflexion, les étudiants proposent des pistes possibles. Ainsi, même s'il n'est pas toujours possible de trouver une solution précise, ils travaillent en groupe pour la faire émerger. Par exemple, lorsque la remarque demande de réduire la complexité ou la profondeur du code, les étudiants concernés proposent des idées générales pouvant améliorer la cible de la remarque, sans pour autant avoir trouvé une solution précise.

Prioritairement, les étudiants accordent de l'importance au fonctionnement. Le critère de qualité passe au second plan. Pour beaucoup, un certain degré de nonchalance se fait ressentir lorsque les remarques concernent l'aspect non fonctionnel du code mais ils cherchent tout de même à trouver une solution.

L'ordre de traitement des remarques a, pour tous les groupes, suivi l'ordre d'affichage. Ils ont parcouru l'ensemble de la liste. L'expérience a démarré sur la vue « liste » pour tout le monde et ils ont tous eu besoin de contextualiser les remarques grâce à la vue de code. Certains faisaient des va-et-vient entre les deux vues disponibles pour accéder à la remarque suivante.

À la fin de l'expérience, les questions posées ont permis de déterminer avec les étudiants quelques critères de priorités de traitement des remarques en situation

normale. Ces réponses sont individuelles, les étudiants ayant chacun répondu aux questions. Les critères qui reviennent le plus souvent sont : correction rapide, remarques qui seraient liées (par exemple à la même fonction, variable), amélioration fonctionnelle, importance de la remarque. Certains seraient décidés à nettoyer la liste pour effacer un maximum de remarques quand d'autres indiquent qu'ils ne s'occuperaient que des remarques qu'ils considèrent comme pertinentes. Ils ont également indiqué évaluer la rapidité de correction via le degré d'importance de la remarque.

6.1.3.2.2. Interface de SonarQube

L'interface de SonarQube est très riche, elle regorge d'informations qu'il n'est pas toujours facile à décrypter. Les étudiants ont eu environ 15 minutes pour se familiariser avec cette interface. Principalement, deux parties de l'interface sont ciblées : la vue de code et le détails d'une remarque.

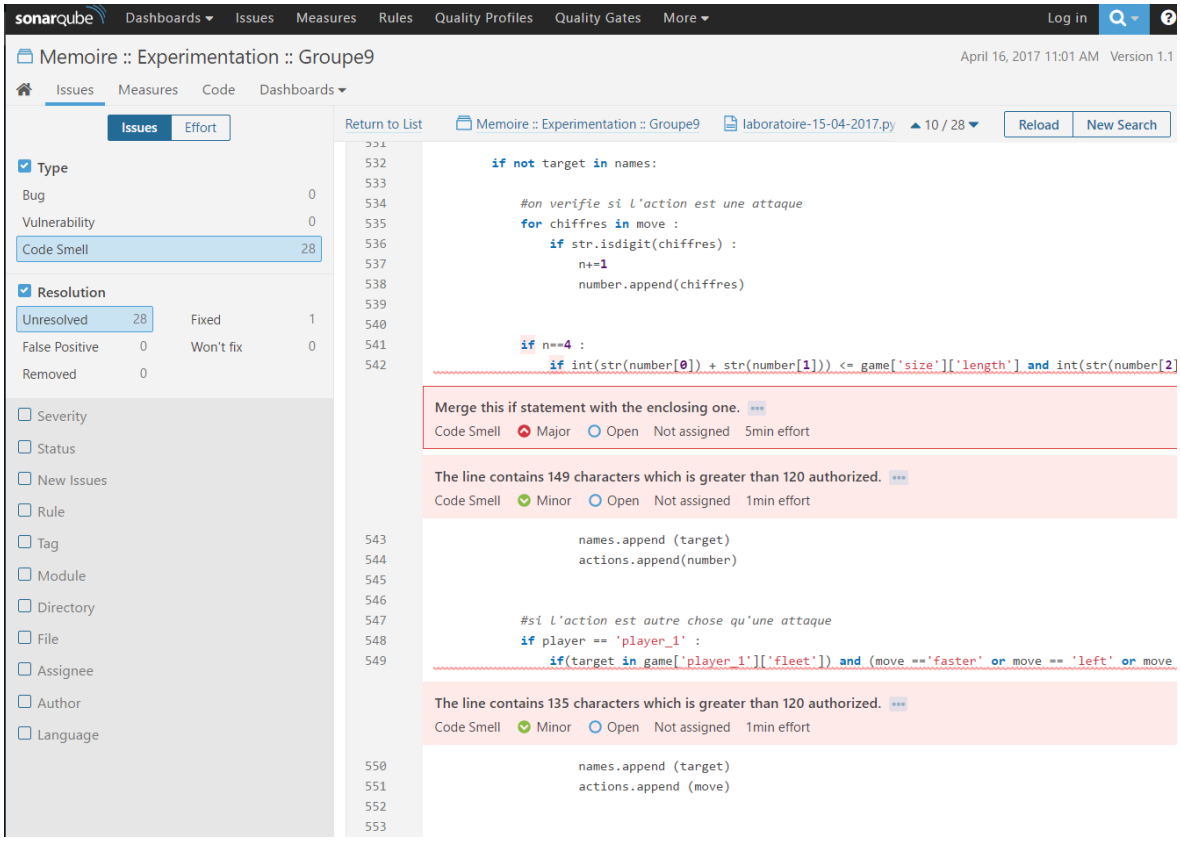


Figure 16: SonarQube - Placement des remarques dans la vue de code : le détail de la remarque sélectionné est encadré de rouge.

Directement lors de l'introduction, les étudiants apprennent que SonarQube offre deux possibilités d'atteindre le code à partir de la liste : soit en sélectionnant la vue dans le menu contextuel, soit en double-cliquant sur la remarque que l'on souhaite localiser. Les étudiants sont satisfaits de voir que SonarQube localise le code automatiquement en plaçant le bloc de la remarque juste sous la ligne concernée. Un soulignement rouge permette de cibler avec précision le code impacté par la

remarque. En supplément, lorsque la remarque est sélectionnée, une couleur de fond, rose pâle, identifie la (les) ligne(s) associée(s). Cependant, certains étudiants ne les remarquent pas et restent parfois bloqués à chercher. La Figure 16 illustre cette description :

- Les lignes 542 et 549 sont cibles de remarques et sont donc soulignées ;
- La remarque sélectionnée fait apparaitre la couleur de fond derrière le « if » présent en ligne 541 et celui présent en ligne 542.

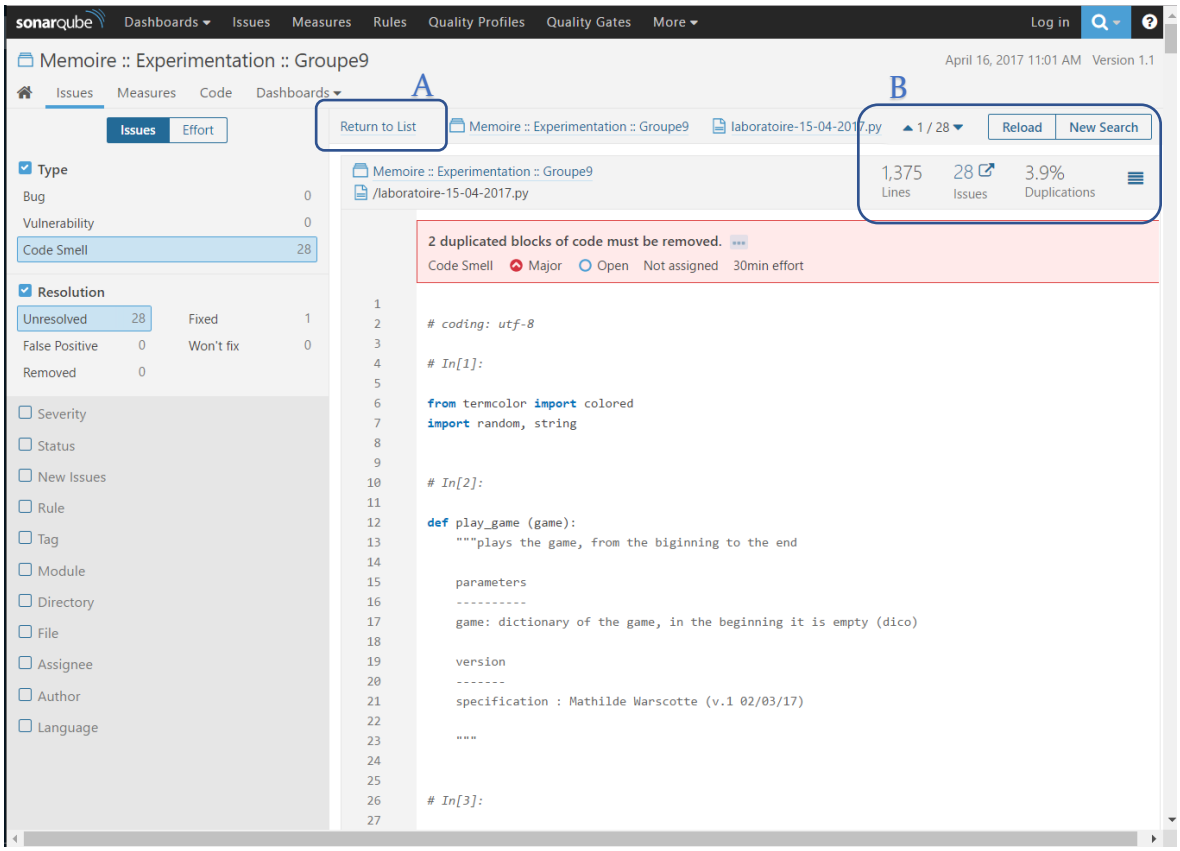


Figure 17: SonarQube - Vue "Code" d'un projet. Groupe A : le lien « retour à la liste » n'est pas assez visible. Groupe B : certaines informations résument l'analyse du code.

Un élément récurrent à tous les groupes a pu être identifié assez rapidement : le bouton de retour à la liste n'est pas assez mis en évidence. Celui-ci est présent sur la vue de code (Figure 17 – Groupe A). Pour tous les groupes qui ont eu besoin de l'utiliser, le problème s'est fait ressentir.

Les étudiants ont, de manière générale, passé peu de temps sur la vue « liste ». La description des détails d'une remarque, lors de l'étape de questions / réponses, s'est d'ailleurs faite, pour la plupart des groupes, à partir de la vue « code ». Pour la majorité des étudiants, les informations indiquées dans ces détails ne sont pas auto-suffisantes. Ils interprètent par exemple la durée indiquée, qui représente la dette technique causée par la remarque, par la durée que prendrait la correction, ou par la durée que l'analyseur a mise pour trouver l'erreur, etc. … Ils ne comprennent pas ce

qui distingue une remarque dite « mineure » d'une remarque dite « majeure » et proposent d'afficher une description afin de rendre la distinction plus claire.

Le classement des remarques ne correspond pas réellement aux estimations des étudiants. Ils ne sont pas tout à fait en accord avec l'importance des remarques. Les plus importantes devraient selon eux d'abord être les remarques qui améliorent le côté fonctionnel du code. Ensuite, dans une moindre mesure, viendraient les critiques sur la syntaxe du code.

Il n'est pas évident de faire le point sur la situation, des étudiants ont émis le besoin d'un résumé visible à partir de la vue de code. Ils n'ont donc pas remarqué la présence de certaines informations en haut de la fenêtre (Figure 17 – Groupe B).

6.1.3.2.3. Utilités de SonarQube

Durant l'expérience, plusieurs groupes ont mentionné la différence d'explicitation entre Canopy, leur environnement de développement, et SonarQube. Il est bien entendu évident que les deux logiciels ont des utilités différentes, mais la complémentarité a été mentionnée. SonarQube donne des informations permettant d'aider à solutionner le problème mentionné. Ils trouvent que cette complémentarité pourrait leur être utile afin qu'ils puissent se construire de bonnes habitudes.

Cette construction de bonnes habitudes passerait par, entre autres : l'uniformisation des noms de variables, même s'ils ne sont pas convaincus que ça va les aider à devenir de meilleurs programmeurs ; l'aide à la détection de code semblable pouvant être factorisé pour une maintenance plus facile ; la réduction de la profondeur de code qui améliore également la complexité.

La majorité des groupes a mentionné une utilité non négligeable de SonarQube : l'apport d'un regard externe au projet. Bien souvent, les étudiants travaillent sur leur propre partie de code, sans que les autres membres ne prennent le temps de relire l'avancement. SonarQube prend donc le rôle d'un membre supplémentaire leur permettant de révéler quelques faiblesses.

Les étudiants ont indiqué leur intérêt à utiliser SonarQube, certains étant néanmoins réticents à l'utiliser seul. Pour cette utilisation, ils risquent cependant de rencontrer quelques problèmes car pour pouvoir exécuter l'analyse, le code doit être syntaxiquement correct. Cela n'a pas toujours été le cas dans le cadre de l'expérience. Il a été nécessaire de commenter certaines parties de code afin de réussir l'analyse.

L'utilisation de SonarQube leur permettrait de rendre leur code plus lisible et plus clair, ceci grâce entre autres à l'aide qu'il apporterait pour améliorer et factoriser le code. De plus, toujours selon eux, il permettrait de trouver des erreurs plus rapidement.

6.2. Conclusions intermédiaires

À la lumière des résultats de cette première expérience, plusieurs conclusions peuvent être tirées.

Tout d'abord, la majeure partie des étudiants ont clairement mis en avant l'utilité de SonarQube, que ce soit pour avoir un regard extérieur, une mise en évidence de faiblesses, pour aider à garder des conventions de nommage, etc. Beaucoup d'entre eux étaient enthousiastes quant à l'utilité de l'outil. Cela témoigne de leur motivation. Cependant, malgré la multitude de fonctionnalités qu'offre SonarQube, aucun étudiant n'a mentionné d'intérêt envers les métriques proposées, les seules pages qui ont été utilisées par les étudiants sont les pages « Liste » et « Code ».

Ensuite, concernant les difficultés de compréhension des remarques que SonarQube souligne, il en résulte une crainte chez certains étudiants s'ils se retrouvaient seuls face à la liste de remarques, ceci à cause d'une faiblesse en anglais. Cependant, lorsqu'ils sont en groupe, la barrière de la langue se fait moins ressentir. En effet, durant l'expérience, aucun groupe n'a été totalement bloqué. Seuls quelques moments ponctuels ont nécessité de l'aide extérieure.

Enfin, la comparaison qualitative de leur code à deux moments séparé d'un contact unique avec SonarQube est encourageante. En effet, 14 groupes sur les 19 impliqués ont réduit le nombre de remarques généré lors de l'analyse.

L'objectif de cette expérience était de valider l'hypothèse de recherche H1 : « *Une sensibilisation des étudiants à la qualité logicielle (à travers l'utilisation d'un outil d'analyse statique) apporte une plus-value dans le contexte particulier de l'apprentissage de la programmation.* » (cfr. Chapitre 3 : Question et Hypothèses de recherche). Les différentes conclusions tirées dans ce chapitre ont mis en lumières des éléments de réponse plutôt positifs.

Chapitre 7 : Préparation de la seconde expérience

Á la suite de l'expérience EXP1 et grâce aux conclusions tirées de celle-ci, il a été décidé de poursuivre le travail et de réaliser l'expérience EXP2. Cependant, afin d'abstraire les aspects techniques inutiles dans le cadre de cette deuxième expérience, il a été décidé de mettre en place un outil : « Code Submitter ». La première partie de ce chapitre consiste en sa description

Ensuite, afin de répondre de manière efficace aux exigences des enseignants, un profil de qualité spécifique a été configuré dans SonarQube. La description des règles utilisées sera réalisée dans la deuxième partie de ce chapitre.

7.1. Code Submitter

7.1.1. Fonctionnalités

Pour permettre aux étudiants d'utiliser SonarQube, il est nécessaire d'abstraire son utilisation au maximum, ceci en créant un outil simple fonctionnant en trois étapes : identification, sélection de fichier et exécution d'analyse.

7.1.1.1. Identification

Dans un souci de confidentialité, l'outil est configuré pour ne fonctionner que lorsque l'utilisateur est connecté. Cela permet de garantir qu'un utilisateur n'est pas en train de soumettre son code chez quelqu'un d'autre. Il faut donc que les différentes pages de l'outil n'acceptent l'utilisateur qu'à la condition qu'il soit authentifié. Si tel n'est pas le cas, l'application redirige l'utilisateur vers la page de login.

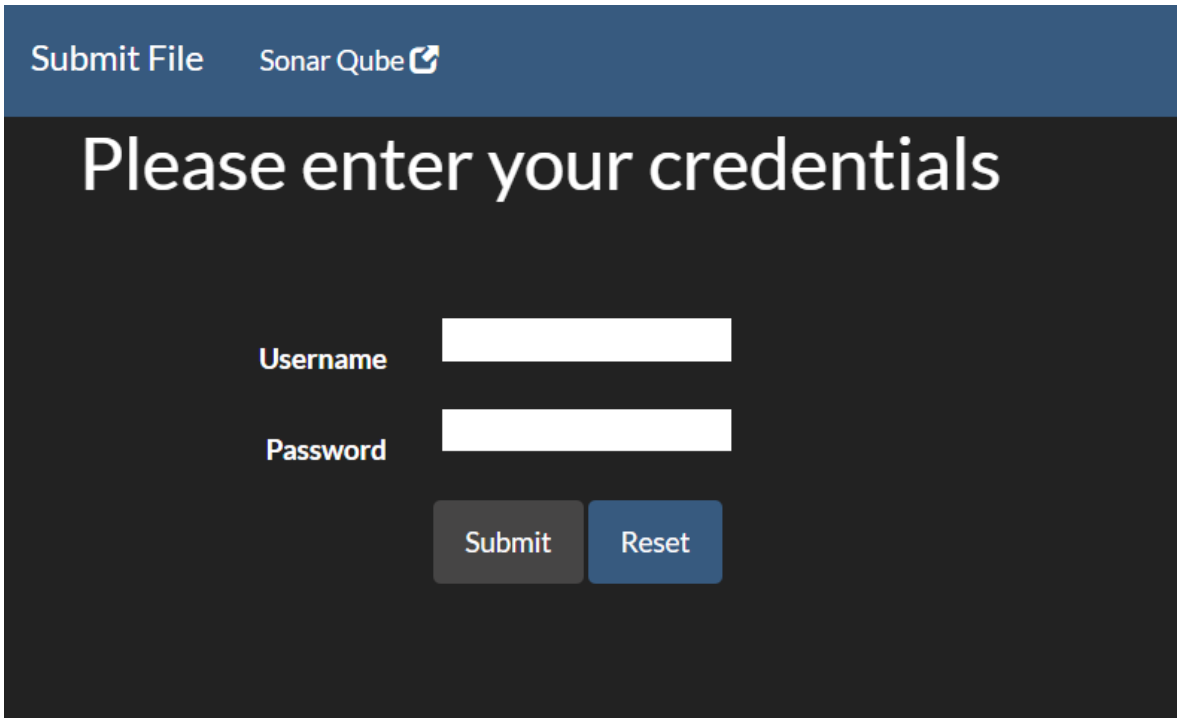


Figure 18: CodeSubmitter - Page de login.

Cette page se compose d'un formulaire de login basique (cfr. Figure 18) : un champ de type texte pour l'identifiant (« UserName »), un champ de type password pour le mot de passe (« Password »), tous les deux requis, et deux boutons (valider et annuler). Le bouton « Reset » vide les champs et le bouton « Submit » lance le processus d'authentification grâce aux informations encodées par l'utilisateur. En cas de réussite, l'application redirige vers la page de soumission de code. En cas d'erreur, la page de login reste affichée garnie d'un message permettant d'en identifier la cause (Figure 19).

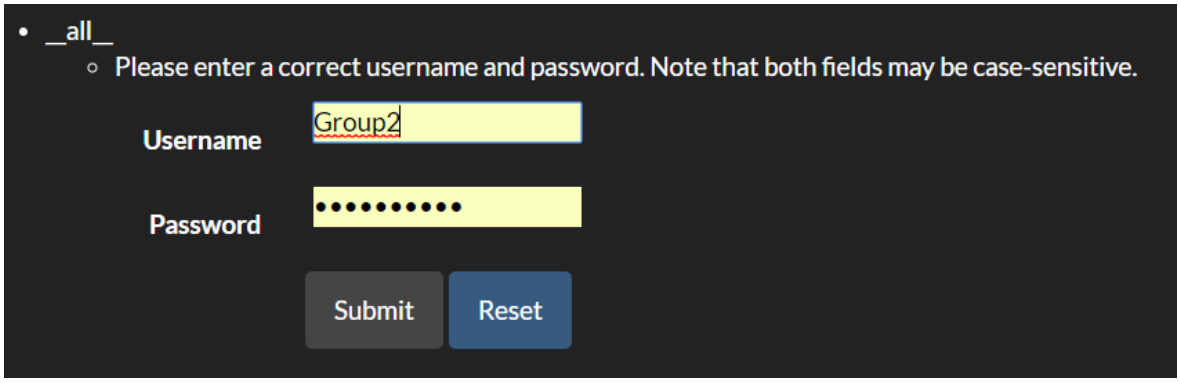


Figure 19: CodeSubmitter - Erreur d'authentification.

Pour simplifier l'utilisation par les étudiants, les mêmes informations de connexion sont utilisées. Le groupe 1 a donc le même identifiant et le même mot de passe sur l'environnement SonarQube et sur l'outil de soumission. L'idée étant de rester simple, l'authentification se concentre sur la connexion. Il n'est pas nécessaire de perdre du temps à développer une fonctionnalité de mot de passe perdu ou d'enregistrement de nouveaux utilisateurs

7.1.1.2. Soumettre le code

La page « Submit File » permet aux utilisateurs de sélectionner un fichier à envoyer pour permettre à SonarQube de l'analyser (Figure 20). Le formulaire contient donc deux champs : le nom du groupe, c'est-à-dire l'utilisateur connecté, et le fichier à envoyer.

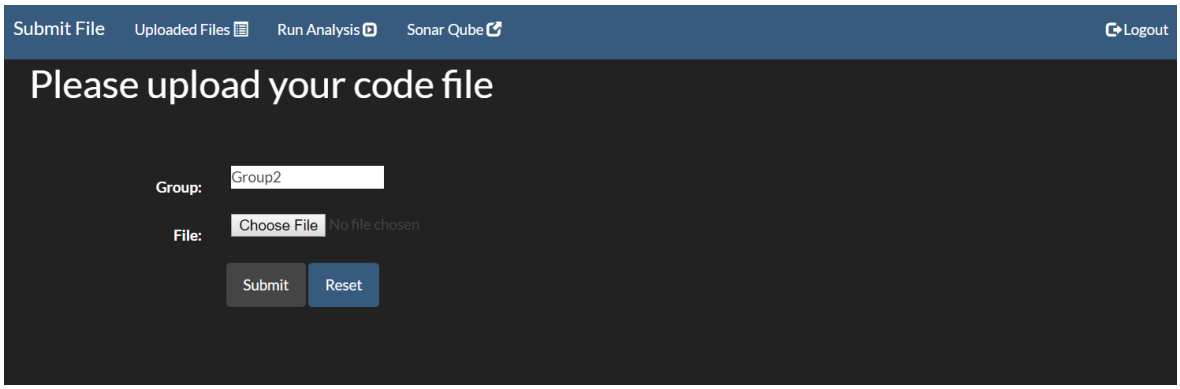


Figure 20: CodeSubmitter - Upload d'un fichier python.

Le fichier s'envoie de manière directe vers la destination adéquate. Il est donc nécessaire de prendre soin du nom de fichier à envoyer. En effet, le système n'effectue pas de vérification d'existence et risque donc d'écraser un fichier existant.

Durant l'utilisation, il a été remarqué que le système était trop permissif. En effet, les étudiants doivent soumettre un fichier de code python afin que SonarQube puisse l'analyser. Or il est arrivé à plusieurs reprises que les étudiants ne transforment pas leur fichier de code, qui est d'origine au format notebook, en format python. Ils sont alors induits en erreur dans SonarQube en ne retrouvant pas leur fichier analysé. Afin de résoudre ce problème, un filtre a été ajouté pour n'accepter que des fichiers au format python. Après ceci, plus aucune plainte n'a été émise.

7.1.1.3. Exécuter l'analyse

La dernière fonctionnalité disponible est l'exécution de l'analyse. Lorsque l'utilisateur clique sur l'option « Run Analysis » du menu, le système exécute de manière synchrone l'analyse de l'ensemble des fichiers qui ont été envoyés jusqu'à présent. Un message, positif (Figure 21) ou négatif (Figure 22), termine l'analyse.

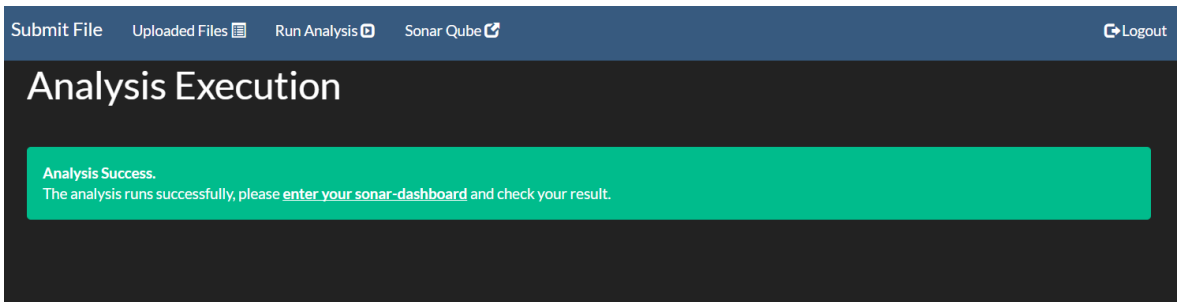


Figure 21: CodeSubmitter - Analyse exécutée avec succès.

Dans le cas où l'analyse s'est déroulée sans problème, il est proposé à l'utilisateur de se rendre sur sa page de projet de SonarQube. Sinon, il lui est demandé de communiquer avec l'administrateur afin de rapporter l'incident.

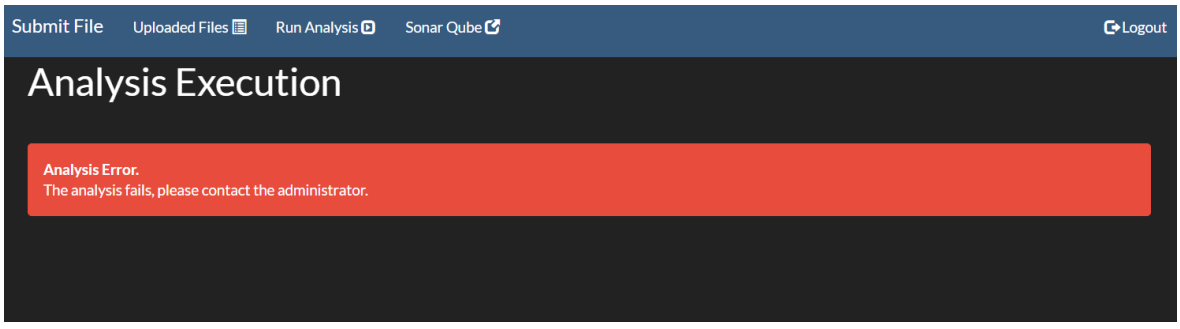


Figure 22: CodeSubmitter - Une erreur est apparue durant l'analyse.

Les liens vers SonarQube sont personnalisés et la redirection s'effectue vers la page du projet. Deux possibilités s'offrent aux étudiants pour accéder à leur page d'accueil de projet : soit ils utilisent l'item de menu qui permet d'accéder à SonarQube à tout

moment de l'utilisation de l'application ; soit ils utilisent le lien présent dans le message de réussite de l'analyse.

7.1.2. Conception

La section précédente analysait les quelques fonctionnalités disponibles de Code Submitter, la suivante traitera de la partie technique. Celle-ci va s'attarder sur la structure finale du projet. L'objectif est ici de décrire (fichiers, répertoires, descriptions, …) et non d'apprendre à créer une application.

7.1.2.1. Langage

Code Submitter est un projet Django, une application web basée sur le langage Python. L'objectif a été de s'intéresser au langage utilisé par les étudiants et évalué lors de ce mémoire.

N'ayant aucune connaissance spécifique à ce langage, il a été nécessaire de l'apprendre durant le développement.

7.1.2.2. Architecture

Le projet est structuré par modules. Chaque module est un répertoire contenant une partie de l'application, une mini-application en soi. De base, le projet en contient un et il est possible d'en ajouter. Celui-ci, « code_submitter », a été utilisé pour configurer l'application. Deux fichiers ont permis de réaliser cette configuration :

1) settings.py pour spécifier le comportement de l'application ;
2) urls.py pour associer à chaque url accessible la vue ciblée.

Une grande partie du fichier settings.py est restée identique à la configuration par défaut. Dans le cadre de l'expérience, il n'est pas nécessaire de changer la base de données, la validation de mots de passe (qui n'a d'ailleurs pas été utilisée), l'internationalisation, etc.

Ponctuellement, il est nécessaire de configurer une variable globale à l'application (adresse de SonarQube, chemin d'accès pour le téléversement de fichier, … etc.). Ces variables sont spécifiées dans ce fichier.

Un deuxième module, Submission, est présent et sert à développer l'application. Comme représenté en Figure 23, celui-ci se compose de :

- un ensemble de fichiers statiques gérés par module dans le répertoire « static » ;
- la définition des vues, le code html, regroupées dans le répertoire templates
- un modèle de données défini au travers du fichier models.py, chaque modification entraine la création de scripts afin de tenir la base de données à jour ; ces scripts sont stockés dans le répertoire migrations;
- un ensemble de formulaires configurés dans le fichier forms.py ;
- une association entre url et vue définie dans le fichier url.py ;

- la définition business des vues dans le fichier views.py.

```
submission
├── migrations
│   └── ...
├── static
│   └── ...
├── templates
│   └── ...
├── models.py
├── forms.py
├── url.py
├── views.py
└── ...
```

Figure 23: CodeSubmitter - structure du module "submission".

La création d'une application Django crée un module d'administration qu'il est possible de redéfinir. Ainsi, la fonction de login / logout peut être utilisée uniquement en redéfinissant le template afin de correspondre à l'interface utilisateur.

7.2. Configuration de SonarQube

SonarQube propose un profil de qualité créé automatiquement, du nom de « Sonar Way ». Cependant, n'étant pas configuré en respectant les exigences des enseignants, il ne parviendrait pas à aider les étudiants dans leur apprentissage de manière optimale. De plus, dans la documentation officielle de SonarQube[15], il est recommandé de créer un profil supplémentaire comprenant l'ensemble des règles à respecter. L'argument de cette suggestion est double : premièrement, « Sonar Way » n'est pas éditable, aucune configuration n'est donc possible ; deuxièmement, lors des mises-à-jour du système, « Sonar Way » est susceptible d'évoluer (ajout de règles, modification de sévérité, etc.). À la lumière de ces arguments, il a été décidé de créer un second profil de qualité, « Profil_BAC1 », et de l'utiliser durant l'expérience EXP2.

Le choix des règles à inclure s'est fait par élimination : l'ensemble d'entre-elles (236 – cfr. Annexe 2) a été présenté à l'enseignant, Monsieur Frénay, qui a sélectionné celles qu'il considérait comme pertinentes. Concernant la configuration des règles, les valeurs par défaut ont été conservées. Le Tableau 1 liste les règles qui ont été incluses dans « Profil_BAC1 ».

[15] https://docs.sonarqube.org/display/SONAR/Quality+Profiles

Règle	Description (Utilisé lorsque ...)
"<>" should not be used to test inequality	La forme “<>” est utilisée pour tester l’inégalité est considéré comme obsolète.
"pass" should not be used needlessly	Le mot clé « pass » est utilisé dans des situations sans utilité. Complément de M. Frénay: “ ‘*pass’ de doit pas être utilisé du tout* ».
Assigning to function call which doesn't return	Une variable est assignée avec le retour d’une fonction et que cette fonction ne contient pas de « return ».
Assigning to function call which only returns None	Une variable est assignée avec le retour d’une fonction et que cette fonction retourne la valeur « None ».
Bad continuation	Une suite de lignes est mal indentée.
Bad indentation	Un nombre inattendu d’indentation ou d’espace est trouvé.
Calling of not callable	Un objet non appelable est appelé.
Collapsible "if" statements should be merged	Deux « if » se succèdent.
Conditions in related "if/elif/else if" statements should not have the same condition	Plusieurs conditions d’un même ensemble de « if », « elif », « else » sont semblables.
Control flow statements "if", "for", "while", "try" and "with" should not be nested too deeply	Les instructions « if », « for », « while », « try », et « with » sont imbriquée à un niveau de profondeur trop important. Cette règle est configurable et la valeur par défaut est de 4.
Duplicate argument name in function definition	Plusieurs paramètres portent le même nom dans la définition d’une fonction.
Duplicate key in dictionary	Un dictionnaire contient plusieurs fois la même clé.
Duplicate keyword argument in function call	Un appel à une fonction se fait en utilisant plusieurs fois le même argument.
Empty docstring	Une classe, une fonction, une méthode ou un module a un docstring vide
Error while code parsing	Une erreur est arrivée lors de la construction de l’arbre syntaxique
Except doesn't do anything	Une clause « Except » ne contient aucune instruction et qu’il n’y a pas de clause « Else ».
Expression is assigned to nothing	Une expression qui n’est pas un appel à une fonction est assignée à rien

Function names should comply with a naming convention	Un nom de fonction ne suit pas les conventions définies. Cette règle est configurable et la valeur par défaut est : « ^[a-z_][a-z0-9_]{2,}$
Functions should not be too complex	La complexité cyclomatique d'une fonction est trop importante. Cette règle est configurable et la valeur par défaut est de 15.
Invalid mode for open	Le mode d'ouverture d'un fichier n'est pas « r », « w », « a » avec « b », « + », « U » comme option.
Invalid name	Le nom ne correspond pas à l'expression régulière définie pour le type associé (une constante, une classe, une variable, etc.).
Line too long	La ligne est trop longue.
Lines should not be too long	La ligne est trop longue. Cette règle est configurable et la valeur par défaut est de 120.
Local variable and function parameter names should comply with a naming convention	Une variable locale ou un paramètre de fonction ne respectent pas l'expression régulière définie. Cette règle est configurable et la valeur par défaut est « ^[_a-z][a-z0-9_]*$
Missing docstring	Une classe, une fonction, une méthode ou un module n'a pas de docstring.
Module imports itself	Un module s'importe lui-même.
More than one statement on a single line	Plusieurs expressions sont situées sur la même ligne.
Multiple values passed for parameter in function call	Un paramètre de fonction est initialisé par deux valeurs : un provenant d'un « positional argument » et un provenant d'un « keyword argument »[16]. Pour rappel, un « positional argument » est un argument qui initialise le paramètre situé à la même position et un « keyword argument » est un argument précédé du nom de paramètre cible.
Nested blocks of code should not be left empty	Un bloc de code est vide et doit être retiré.
Not enough arguments for format string	Une chaine de caractère avec des paramètres ne contient pas assez d'arguments.

[16] https://docs.python.org/2/glossary.html#term-argument

Parser failure	Le parser de Python n'arrive pas à parser le fichier.
Sections of code should not be "commented out"	Des sections de code ont été commentées. Elles sont à retirer car peuvent être récupérées du « source control ».
Similar lines	Un ensemble de lignes similaires ont été détectées. Cela indique que le code peut être factorisé.
Source files should have a sufficient density of comment lines	Un fichier de code ne contient pas assez de commentaire pour atteindre un seuil de densité. La règle est configurable et le paramètre par défaut est de 25.
Source files should not have any duplicated blocks	Un bloc de code dupliqué a été détecté dans le fichier.
Too few arguments	Un appel à une fonction contient trop peu d'argument.
Too many arguments	Un appel à une fonction contient trop d'argument(s).
Too many branches	Une fonction ou une méthode contient trop de branchements, la rendant trop complexe à suivre.
Too many local variables	Une fonction contient trop de variables locales.
Undefined name	Un nom ne peut être retrouvé dans un module (pas de variable, de fonction, etc.).
Undefined variable	Une variable non encore définie est accédée.
Unnecessary parentheses	Une condition ne contient qu'un seul élément à l'intérieur des parenthèses pour un « if », un « for », etc.
Unreachable code	Du code se trouve après une instruction « return » ou « raise ».
Unused variable	Une variable est définie mais jamais utilisée.
Using variable before assignment	Une variable est accédée avant d'avoir été assignée

Tableau 1: Liste des règles utilisées pour la réalisation de l'expérience EXP2 : la colonne de gauche cite la règle quand la colonne de droite la décrit / traduit

7.2.1. Configuration des utilisateurs

Afin de permettre aux étudiants d'accéder aux résultats des analyses de leur code, il est nécessaire de leur permettre d'accéder à SonarQube. Deux possibilités sont à envisager.

Premièrement, il est nécessaire de laisser les projets publics afin que tout le monde puisse y accéder. Il suffit alors aux étudiants de sélectionner leur projet parmi la liste complète. L'avantage principal de cette solution est sa facilité de réalisation. En effet, lorsqu'un projet est analysé, celui-ci est créé dans SonarQube et les paramètres de confidentialité par défaut s'y appliquent. Toute personne accédant à l'interface de SonarQube reçoit donc l'accès au projet. Le principal avantage est également le principal problème. En effet, comme explicité précédemment, pour chaque projet nouvellement créé, tous les utilisateurs y ont accès, le code est donc rendu public.

La seconde possibilité consiste en la création d'utilisateurs et l'application de paramètres de confidentialité spécifiques à chaque projet. Il est en effet possible de créer un utilisateur, d'analyser un projet vide afin de lui créer une entrée dans SonarQube et de modifier les paramètres de confidentialités associés à ce nouveau projet afin d'autoriser uniquement le nouvel utilisateur à y accéder. À l'inverse de la première solution, le principal avantage se situe ici dans la confidentialité. En effet, chaque utilisateur a accès uniquement à son projet. En ce qui concerne la mise en place, celle-ci est plus longue.

Dans ce contexte pédagogique, il serait inconcevable de laisser le libre accès aux étudiants pour tous les projets. Chaque groupe est responsable de son projet et le plagiat est interdit. La seconde solution s'impose donc grâce à son argument de confidentialité.

D'un point de vue pratique, le nombre d'étudiants étant relativement élevé (environ une centaine), il est donc nécessaire de trouver une solution alternative permettant de réduire le nombre d'utilisateur à gérer. La solution idéale est de créer un utilisateur par groupe d'étudiants (environ).

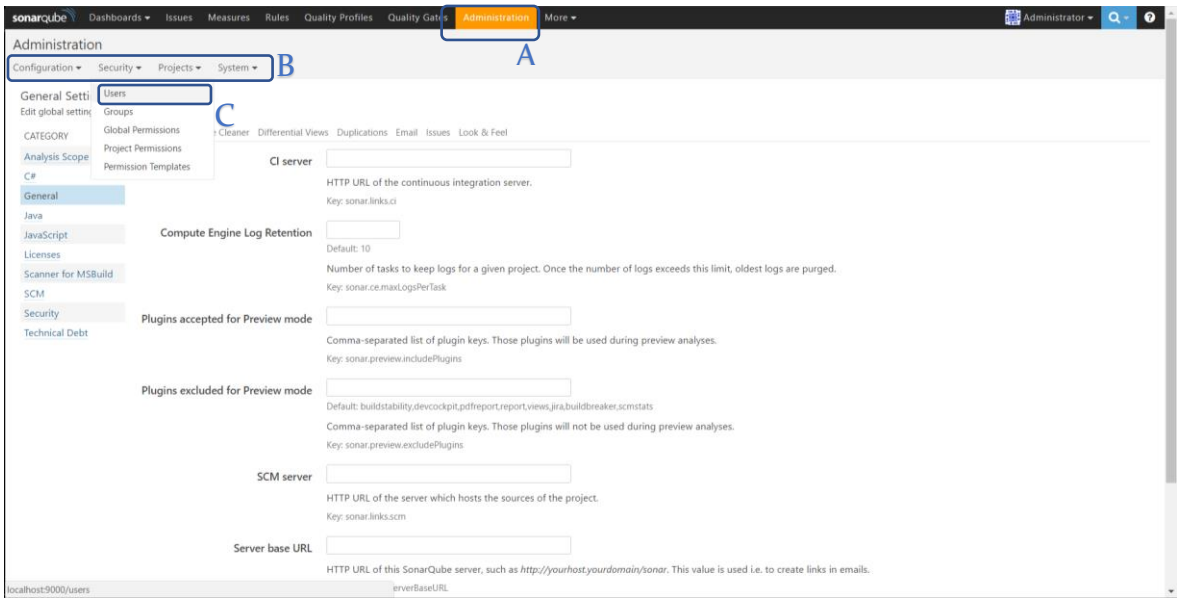


Figure 24: SonarQube – Administration. Groupe A : Item de menu permettant d'accéder à la page. Groupe B : sous-menu de la fonctionnalité « Administration ». Groupe C : Item de sous-menu « Security > Users » permettant d'accéder à la gestion des utilisateurs de l'instance courante de SonarQube.

La création d'un utilisateur s'effectue en quelques clics. Dans un premier temps, la page « *Administration* » (Figure 24), accessible via le menu supérieur (Groupe A), permet d'accéder à l'ensemble des réglages de l'application. De la même manière que dans les projets, cette page contient un menu contextuel (Groupe B). Dans la catégorie « *Security* » est présent l'option « *Users* » (Groupe C). Cette page permet la gestion des utilisateurs.

La page des utilisateurs est une liste basique permettant d'identifier les informations importantes rapidement. Le bouton « *Create User* » permet d'ajouter un nouvel utilisateur.

Après avoir rempli les champs nécessaires et cliqué sur le bouton « *Create* », l'administrateur est redirigé vers la liste mise à-jour (Figure 25). À partir de cette vue, l'utilisateur « *Groupe1* » (Groupe A) peut être assigné à un ou plusieurs groupes d'utilisateur, peut être bloqué, modifié ou encore supprimé.

Après avoir créé les utilisateurs nécessaires, il est nécessaire de les assigner à leur projet respectif. Pour cela, le menu contextuel contient une entrée « *Projects* » (Groupe B) contenant un item « *Management* » (Groupe C). La vue des projets est similaire à la vue des utilisateurs, l'action de création est en tout point semblable. La liste des projets contient, pour chacun d'entre eux, un lien vers la page de détails. Celle-ci est bien entendu vide. Les droits d'administrateur offrent au menu contextuel l'entrée « *Administration* ». Sous cette entrée est présente l'item « *Permission* ».

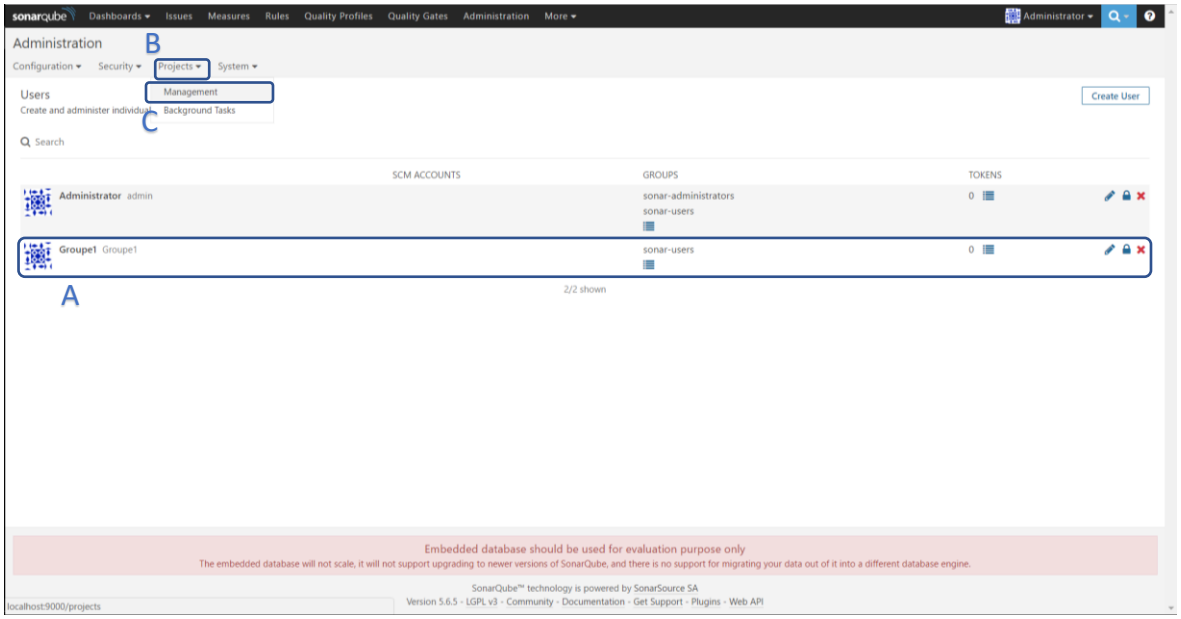


Figure 25: SonarQube - Liste des utilisateurs. Groupe A : un utilisateur parmi la liste. Groupe B : item de menu « Projects ». Groupe C : item « Management » du sous-menu « Projects ».

Il est possible de gérer 5 permissions différentes associées au projet (Figure 26) :

- « *Browse* » : permet d'accéder à un projet, d'accéder aux mesures et de gérer les remarques ;
- « *See Source Code* » : permet d'accéder à la vue « *Code* » et nécessite la permission « *Browse* » ;
- « *Administer Issues* » : permet de manipuler les remarques et nécessite la permission « *Browse* » ;
- « *Administer* » : fait apparaître l'entrée « *Administration* » dans le menu contextuel ; nécessite la permission « *Browse* » ;
- « *Execute Analysis* » : permet de gérer les analyses sur le serveur ;

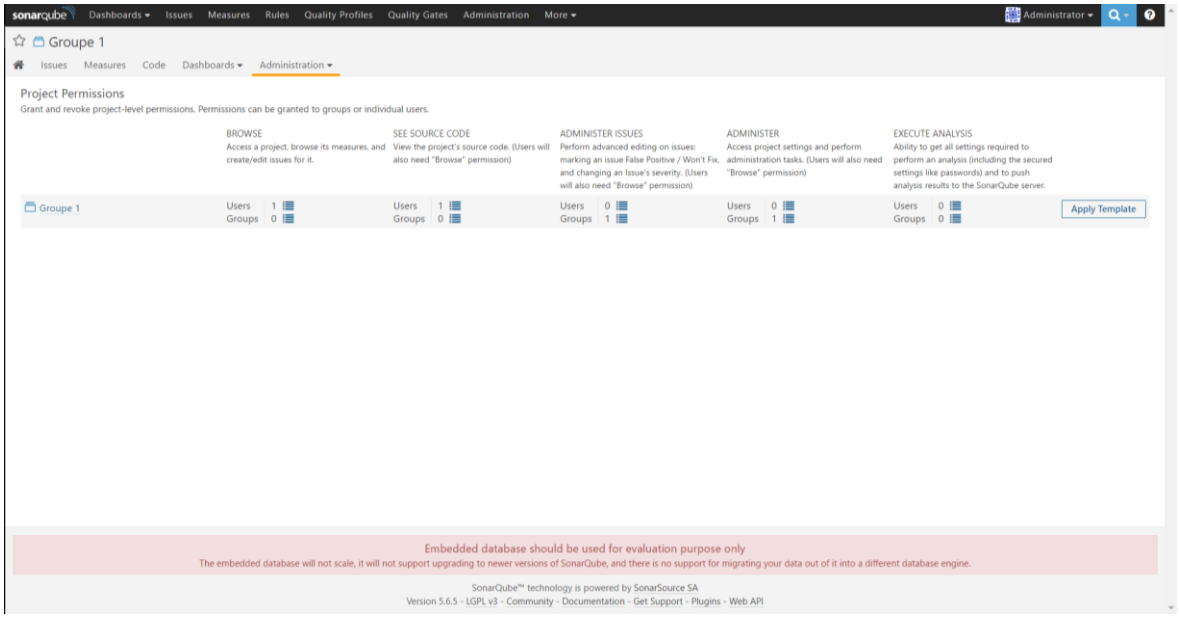


Figure 26: SonarQube - Permissions associées à un projet. L'ensemble des projets est affiché sur la même page (ici un seul projet est présent : « Groupe 1 »). Cinq permissions sont configurables, « Browse », « See Source Code », « Administer Issues», « Administer », « Execute Analysis ».

Comme spécifié précédemment, la création d'un nouveau projet applique les paramètres de confidentialité par défaut. La permission « *Browse* » est disponible pour tout le monde ainsi que « *See Source Code* ». « *Administer Issues* » et « *Administer* » quant à elles sont réservées aux administrateurs. « *Execute Analysis* » n'est attribuée à personne. Pour la configuration finale, il faut retirer « tout le monde » des deux premières permissions et y ajouter l'utilisateur cible.

De cette manière, les utilisateurs non présents dans la liste de la permission « *browse* » n'ont pas accès à ce projet. La configuration finale leur donnera uniquement accès au leur. Il leur est impossible d'accéder au code d'un autre groupe.

7.2.2 Configuration du profil de qualité

Un profil de qualité est un ensemble de règles à vérifier lors de l'analyse. Logiquement, tous les projets d'un même langage devraient être analysés par le même profil de qualité. Mais il peut arriver que certains projets spécifiques demandent une attention particulière ou à l'opposé, que certains projets développés dans une technologie différentes requièrent une exigence moindre.

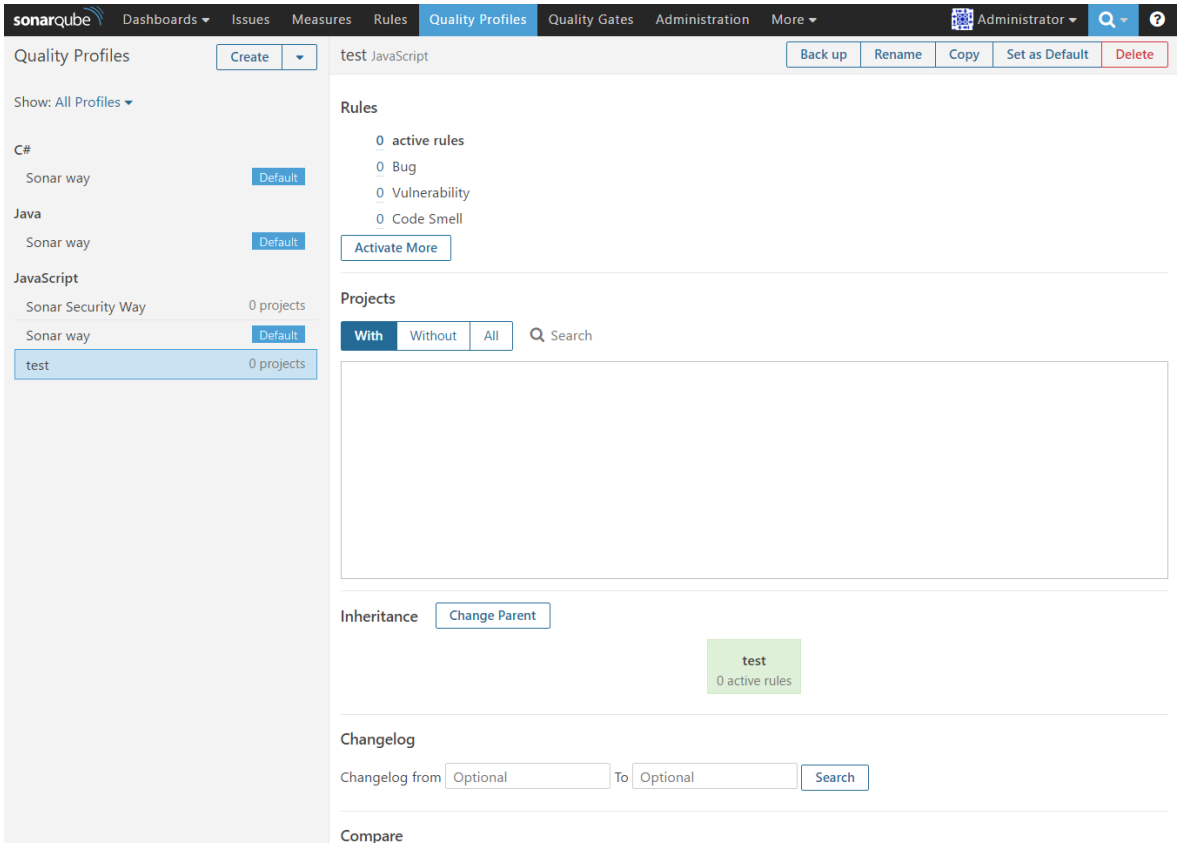


Figure 27: SonarQube - Liste des profils de qualité. La partie de gauche liste l'ensemble des profils de qualité groupés par langage. La partie de droite résume celui sélectionné.

SonarQube propose un profil par défaut. Il est donc facile de gérer les premières analyses. La création d'un nouveau qui est plus adapté aux besoins se réalise en trois étapes.

Premièrement, il est nécessaire de créer le profil de manière effective. Le menu contient une entrée « *Quality Profiles* ». La page de destination (Figure 27) contient un bouton « *Create* ». Au clic sur celui-ci, un formulaire apparait demandant les informations de base : le nom du profil et le langage.

Le profil est créé mais ne contient encore aucune règle. S'il devait être utilisé pour analyser un projet, il ne déclarerait aucune remarque. La deuxième étape consiste en l'ajout de règles dans le profil. Cette action peut être réalisée grâce au bouton « *Activate More* ». Celui-ci dirige vers la liste des règles existantes (Figure 28). Chaque élément de la liste contient un bouton contextuel permettant d'activer ou de désactiver la règle dans le profil sélectionné.

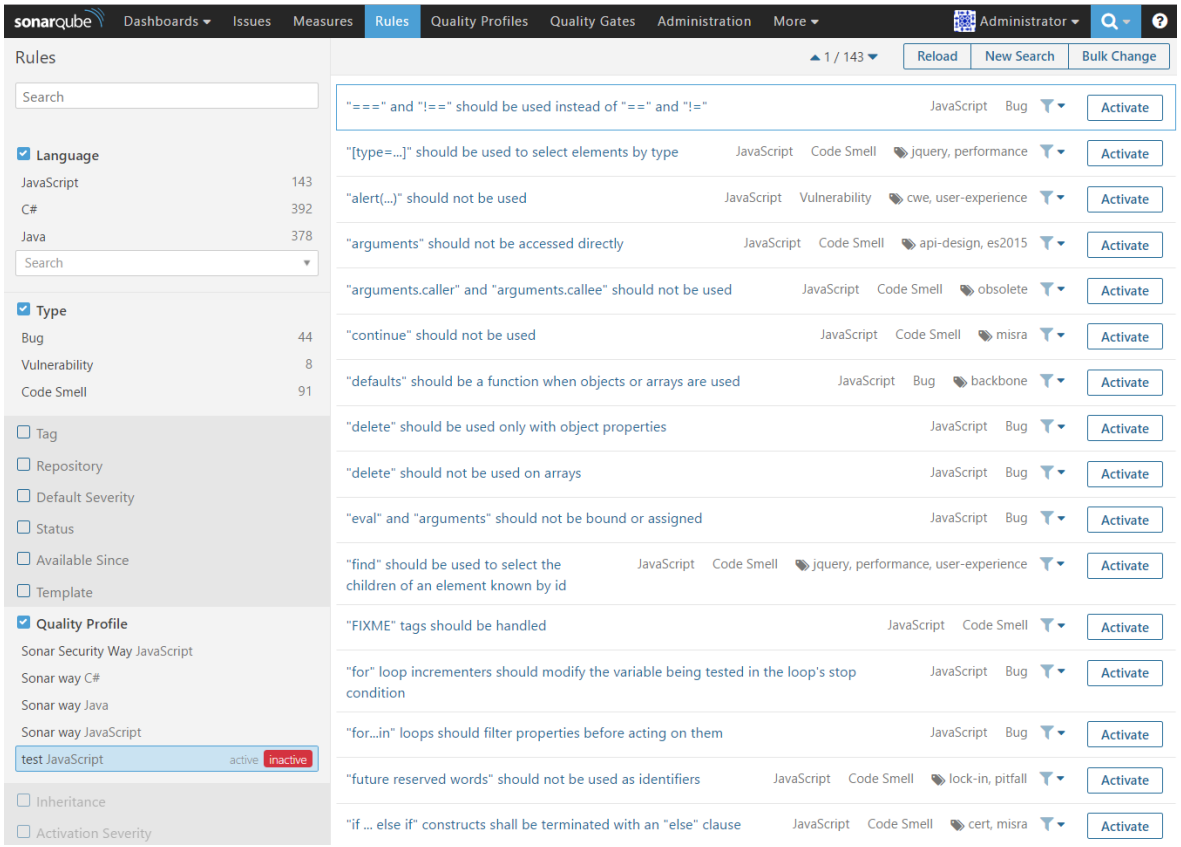


Figure 28: SonarQube - Liste des règles filtrées pour ne faire apparaitre que celles présentes dans le profil de qualité « test ».

La dernière étape nécessaire pour associer le profil nouvellement créé avec un projet à analyser. Ceci se réalise au travers du groupe « *Projects* » dans la vue du profil.

Chapitre 8 : Seconde expérience – En route vers l'autonomie

La seconde expérience, EXP2, a été mise en place afin d'offrir l'environnement de SonarQube aux étudiants pour une longue période. L'objectif de EXP2 est de répondre à l'hypothèse de recherche H2 : « *L'outil d'analyse statique choisi dans le cadre de ce mémoire est utilisable, sans aide extérieure, par les étudiants* ».

8.1. Données collectées

Afin de garantir aux étudiants les connaissances nécessaires pour utiliser l'outil de soumission de code, « Code_Submitter », un guide (cfr. Annexe 3) a été créé et distribué à chacun des groupes. De plus, un fichier au format PDF a été mis à disposition sur WebCampus. En plus de ceci, lors de l'annonce de la mise à disposition de SonarQube, une démonstration en direct a eu lieu : l'auteur a parcouru le guide en réalisant les étapes sur un ordinateur modèle[17] et a terminé en répondant à d'éventuelles questions provenant des étudiants. Cette manipulation a eu lieu le vendredi 24 mars. Grâce à ces compléments d'informations, les étudiants ont toutes les cartes en main pour pouvoir utiliser SonarQube à leur guise. Si malgré tout, il leur arrivait de se poser une question, de rencontrer un problème, etc. les étudiants auraient la possibilité de demander de l'aide aux assistants durant les séances tutorées du vendredi.

Les systèmes (SonarQube et Code_Submitter) ont été déployés sur un environnement interne à l'UNamur. Pour la plupart des environnements rendus disponibles aux étudiants, la politique interne de l'UNamur est de les laisser accessibles uniquement en interne au réseau. Lors de l'installation des deux logiciels, leurs configurations ont rendu SonarQube atteignable de l'extérieur alors que Code_Submitter ne l'était pas. Lorsque les étudiants n'étaient pas à l'université, ils ne pouvaient donc pas lancer de nouvelles analyses mais pouvaient tout de même consulter les résultats. Durant les six semaines de disponibilité, les étudiants n'ont donc pas eu accès à Code_Submitter durant six weekends et les deux semaines de vacances de Pâques.

L'expérience a produit un unique type de donnée permettant de valider l'hypothèse H2 : un historique d'utilisation de SonarQube. Celui-ci a été transformé en tableau et peut être consulté via la Figure 29. Dans ce tableau, les jours non-scolaires ont été teintés de gris, les jours de week end sont restés vides et, pour les jours sans utilisation, la couleur de la police a été atténuée afin de faire ressortir les jours importants. À la suite de ce tableau a été ajouté une ligne résumant le nombre d'utilisations totale par groupe. Enfin, un code couleur permet de repérer les différents totaux : pour deux utilisations ou moins un dégradé de rouge est utilisé, pour trois utilisations le fond reste blanc et pour plus de trois utilisations un dégradé de vert est utilisé.

[17] Plusieurs écrans sont répartis dans le local où les étudiants ont cours.

	Groupe 1	Groupe 2	Groupe 3	Groupe 4	Groupe 5	Groupe 8	Groupe 9	Groupe 10	Groupe 12	Groupe 13	Groupe 14	Groupe 15	Groupe 16	Groupe 18	Groupe 20	Groupe 21	Groupe 23	Groupe 24	Groupe 26	Groupe 30	Groupe 31	Groupe 41	Groupe 42
24-mars	1	1	0	2	3	0	1	2	0	1	3	1	1	3	6	0	0	3	1	1	0	0	0
25-mars																							
26-mars																							
27-mars	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
28-mars	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
29-mars	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
30-mars	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
31-mars	0	0	0	0	1	2	0	0	0	1	0	5	0	0	0	0	0	0	6	0	1	0	0
01-avr																							
02-avr																							
03-avr	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
04-avr	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
05-avr	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
06-avr	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
07-avr	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
08-avr																							
09-avr																							
10-avr	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11-avr	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12-avr	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
13-avr	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14-avr	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
15-avr																							
16-avr																							
17-avr	0	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18-avr	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
19-avr	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
20-avr	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0
21-avr	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0
22-avr																							
23-avr																							
24-avr	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
25-avr	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
26-avr	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
27-avr	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
28-avr	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
29-avr																							
30-avr																							
01-mai	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
02-mai	0	0	0	0	0	0	0	0	0	2	1	0	0	0	0	0	0	0	0	0	0	0	0
03-mai	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
04-mai	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
05-mai	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	2	0	0	0	0	0	0	0
Utilisations totales	5	6	0	3	4	2	1	2	1	4	5	6	4	4	7	2	1	3	7	1	2	0	3

Figure 29: EXP2 - Historique d'utilisations de SonarQube. Liste le nombre d'utilisations, par jour, de SonarQube par les étudiants durant la période de disponibilité (24 mars -> 5 mai). La dernière ligne totalise celles-ci. Les jours non-scolaires sont grisés et la date de remise du projet est colorée de vert.

8.2. Analyse et discussion des résultats

L'analyse des résultats va cibler deux aspects : le nombre d'utilisations et les périodes auxquelles les étudiants ont utilisé les outils.

8.2.3. Analyse du nombre d'utilisations

En six semaines d'utilisation, répartis sur 23 groupes, les étudiants ont, au total, utilisé SonarQube à 73 reprises, ce qui équivaut à une moyenne par groupe de 3,17 fois. En portant l'attention sur les statistiques quotidiennes, il est intéressant de constater que plusieurs groupes ont effectué des « grappes » d'analyses, c'est-à-dire plusieurs par jour. Par exemple, le Groupe 20 qui en totalise sept en a effectué six lors du premier jour de disponibilité. Donc au lieu de considérer six soumissions, il serait plus intéressant de considérer que le jour est actif.

Cependant, les étudiants ne pouvant être tous actifs chaque jour dans la réalisation de leur projet, il serait donc, en définitive, plus utile d'élargir la période d'activité à une semaine pour obtenir un délai représentatif. Sur les six semaines disponibles, deux étaient des vacances et donc pour beaucoup d'étudiants, il n'est pas possible de se rendre à l'Unamur. Il reste alors 4 semaines pendant lesquelles ils peuvent soumettre des versions intermédiaires de leur code. Un maximum de deux semaines ont été actives et ce pour quatre groupes. 13 groupes ont, quant à eux, étés actifs durant une seule semaine. Les derniers groupes n'ont participés à aucune des semaines de disponibilité.

N'ayant pas défini d'objectif de participation pour l'expérience, il est assez difficile de juger. Cependant, lors de l'expérience EXP1, les étudiants semblaient très enthousiastes quant à une utilisation de SonarQube pour leurs projets (cfr. Chapitre 6 : Première expérience – Une utilisation « observée » de l'outil). Il est évident que le nombre de participations est faible. Plusieurs raisons peuvent l'expliquer.

Premièrement, comme explicité précédemment, l'environnement est disponible uniquement en interne. Il est obligatoire d'être connecté au réseau de l'Unamur pour pouvoir accéder à Code_Submitter. L'accès au système est limité, voire impossible, dans plusieurs situations : pour les étudiants qui travaillent sur leur projet chez eux, les week end et les jours de vacances.

Deuxièmement, suite aux différents retours d'étudiants récoltés durant l'expérience, il a été possible de construire un avis indicatif concernant les résultats de leurs analyses. Pour certains d'entre eux, l'analyse n'était pas suffisamment pertinente. En effet, de l'analyse de leur fichier de code ressortaient des remarques qu'il ne leur était pas possible de corriger. Pour exemple, les règles « Line too long » et « Lines should not be too long » sont enfreintes lorsque les étudiants codent l'affichage d'une ligne dans la console. Ils se retrouvent alors avec un nombre dérangeant de remarques qu'ils considèrent comme inutiles et qu'ils ne corrigeront pas. SonarQube perd donc en crédibilité auprès des étudiants.

Troisièmement, les étudiants favorisant l'aspect fonctionnel de leur logiciel, ils n'apportent que peu d'intérêt au côté « esthétique » du code et l'utilisation de SonarQube passe donc en second plan.

Pour terminer, il s'agit d'une expérience externe à leur projet, l'utilisation de SonarQube, ou non, n'impacte en rien leur évaluation. Il n'est pas fait mention de ce système dans leur énoncé, tous les assistants ne sont pas informés de cette initiative, etc. Bref, l'utilisation de SonarQube n'est pas vraiment intégrée à leur développement.

8.2.2. Analyse des périodes d'utilisations

Comme expliqué ci-dessus, les étudiants ont des séances tutorées tous les vendredis. Il peut donc paraitre évident que les soumissions se regroupent durant ces jours où les assistants sont présents et disponibles pour les aider. Cette tendance se vérifie lors de la première semaine de disponibilité de Code_Submitter. En effet, six groupes ont soumis un fichier de code à analyser le vendredi 31 mars. Il s'agit de la plus haute fréquentation journalière de toute l'expérience. L'attrait des étudiants envers SonarQube pour ce jour particulier peut en partie s'expliquer par la présence des deux semaines de vacances de Pâques du lundi 3 avril au vendredi 14 avril.

Durant ces vacances, donc malgré l'absence de cours, un groupe a été actif le vendredi 7 avril. Cela semble étonnant et confirme donc de l'intérêt de certains étudiants envers SonarQube (cfr. Chapitre 6 : Première expérience – Une utilisation « observée » de l'outil).

Lors de la semaine suivant les vacances, sept groupes ont été actifs ; il s'agit de la plus haute fréquentation hebdomadaire. Une autre caractéristique la différencie des premières semaines d'utilisation : les soumissions sont éparpillées. Les étudiants n'ont pas attendus la séance tutorée du vendredi. Il s'agit d'un second témoignage de l'intérêt de certains, autres, groupes de leur intérêt envers SonarQube.

Pendant la dernière semaine, les étudiants ont rendu leur projet le 3 mai. Deux groupes ont trouvé utile de réaliser une ultime analyse avant la remise ; cela peut paraitre normal, ils désiraient certainement s'assurer de la qualité de leur code avant de le valider. Les outils étant disponibles après la date de fin de projet, il a été observé trois groupes actifs durant cette période, les jeudi 4 et vendredi 5 mai. Ceci peut s'expliquer par l'envie des étudiants de continuer d'améliorer leur programme. En effet, il est offert aux étudiants la possibilité de tester entre eux leurs programmes dans des situations de confrontation.

8.3. Conclusions

L'objectif de cette expérience était de valider l'hypothèse de recherche H2 : « *L'outil d'analyse statique choisi dans le cadre de ce mémoire est utilisable, sans aide extérieure, par les étudiants.* » (cfr. Chapitre 3 : Question et Hypothèses de recherche). Ce chapitre a permis de tirer plusieurs conclusions, certaines négatives telles que le nombre d'analyses de code réalisé par les étudiants, et certaines positives telles que l'autonomie

que certains groupes ont montrés face aux outils. De plus, 17 groupes ont été actifs au moins une fois durant la disponibilité et valident donc l'intérêt mentionné durant la première expérience. Ces différentes conclusions semblent donc positives quant à la validation de l'hypothèse H2.

Chapitre 9 : Conclusion

9.1. Résultats

Initialement, le mémoire avait pour but de tenter de répondre à la question suivante : « *L'utilisation d'un outil d'analyse statique de code permet-il à des novices d'améliorer leur apprentissage de la programmation ?* » (cfr. Chapitre 3 : Question et Hypothèses de recherche). Deux hypothèses ont été traitées au travers de ce travail afin d'apporter des éléments de réponse :

- H1 : Une sensibilisation des étudiants à la qualité logicielle (à travers l'utilisation d'un outil d'analyse statique) apporte une plus-value dans le contexte particulier de l'apprentissage de la programmation.
- H2 : Un outil d'analyse statique est utilisable, sans aide extérieure, par des novices.

Pour y parvenir, deux expériences ont été mise-en-place durant l'année scolaire (cfr. Chapitre 5 : Méthodologie de recherche) :

- EXP1 a consisté à mettre les étudiants en contact avec l'outil d'analyse statique choisi, SonarQube, et à observer leur comportement face à celui-ci.
- EXP2 a mis cet outil à disposition des étudiants durant six semaines. Ceci s'est fait au travers d'un environnement personnalisé.

EXP1, au travers des trois types de données récoltées durant sa réalisation (séquences vidéo du passage de chaque groupes, résultats de l'analyse par SonarQube du code des étudiants à deux moments différents et réponses au questionnaire d'après séance), a mis en avant plusieurs éléments qui ont permis de guider la suite du travail à réaliser pour ce mémoire.

Tout d'abord, l'analyse qualitative des séquences vidéo a permis de mettre en avant la méthodologie mise en place par les étudiants pour tenter de corriger les remarques soulignées par SonarQube. Celle-ci passe par trois étapes : la lecture / traduction, l'interprétation et enfin la recherche de solution. Durant l'étape de lecture / traduction, la majeure partie des groupes ont montré une faiblesse en anglais, ce qui pourrait devenir handicapant en cas d'utilisation seul. L'étape d'interprétation a mis en évidence deux types de remarques, les remarques « évidentes » qui donne la solution dans la description et les remarques « complexes » qui nécessitent une compréhension et une contextualisation afin d'être capable de construire une solution. Les vidéos ont également montré les difficultés rencontrées par les étudiants quant à l'utilisation de SonarQube. En effet, l'interface semble remplie d'informations, ce qui a tendance à noyer les informations utiles. Un dernier élément important a été identifié : l'intérêt que les étudiants portent envers SonarQube. Ils semblent presque tous prêt à l'utiliser durant les laboratoires.

Ensuite, grâce à deux résultats d'analyse de SonarQube, l'une réalisée pour l'expérience et l'autre réalisée avec le code remis pour l'évaluation finale de leur mini-projet, il a été possible de comparer de manière quantitative les différentes règles violées. Cette comparaison s'avère très positive car 14 groupes sur les 19 impliqués ont réussi à s'améliorer en réduisant le nombre de remarques.

Et enfin, grâce aux réponses aux questionnaires d'après séance il a été possible de déterminer que les étudiants sont intéressés par l'utilisation d'outils tels que SonarQube. En effet, ils semblent particulièrement attirés par l'aspect « résumé » que peut apporter ce genre d'outils et donc de faire gagner un certain temps dans leur travail tout en apportant un regard externe.

EXP2, avec uniquement l'historique de soumission des étudiants, a mis en avant le faible nombre d'utilisations des étudiants lorsque SonarQube est libre d'accès, ce qui peut s'expliquer par plusieurs raisons telles que la mauvaise configuration du profil de qualité de SonarQube, l'absence d'intégration dans le processus de développement des étudiants, etc. Cependant, la majeure partie des groupes ont utilisé, au moins une fois, SonarQube ; pour certains à une période inattendue (durant les vacances, après la remise de leur projet, …), ce qui témoigne de leur intérêt envers cet outil.

L'objectif du mémoire était de valider, ou non, les hypothèses de départ. EXP1 a permis de valider H1, en effet, en étant soutenus, aidés par moment et recadrés, les étudiants ont montré des signes encourageants concernant leur niveau d'apprentissage de la programmation. EXP2 a été initialement construite afin de valider H2. Cependant, plusieurs éléments portent à croire que SonarQube n'est pas un outil qui convient à des novices. Plusieurs groupes d'étudiants l'ont utilisé durant les six semaines de disponibilité mais un nombre trop peu de fois et de manière trop irrégulière. SonarQube semble donc être un outil efficace mais inadapté aux novices.

9.2. Réflexions futures

Afin de pallier aux différentes faiblesses rencontrées durant la réalisation de l'expérience EXP2, différentes pistes peuvent être envisagées.

Tout d'abord, il pourrait être intéressant de ne pas proposer SonarQube en accès libre mais de le rendre disponible lors d'échéances pré-établies. Celui-ci n'étant pas inclus dans leur cours de programmation, il est évident que ce type d'outil ne fait pas partie du cycle de développement habituel des étudiants. Les échéances serviraient alors de rappel pour son utilisation.

Ensuite, en ce qui concerne le profil de qualité configuré, il s'est avéré être trop exhaustif, peut-être serait-il utile de se concentrer sur certaines notions telles que la factorisation ou la simplification au lieu de tenter de tout regrouper pour avoir une évaluation globale.

Enfin, SonarQube a été utilisé tel-quel, c'est-à-dire que son interface n'a pas été retravaillée. Or il s'agit d'un outil conçu pour professionnels, qui possèdent un background technique conséquent. Il serait dès lors intéressant de la simplifier, de retirer les éléments juger non-pertinents dans le cadre du cours afin de mettre en évidence ceux qui, à l'opposé, méritent toute l'attention.

Bibliographie

Ataíde, A. (2014). *Static Code Analisys.* ACM.

Becker, B. A. (2016). An effective approach to enhancing compiler error messages. *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pp. 126 - 131.

Conseil Wallon de la Politique Scientifique. (2013). *Attractivité des études et métiers scientifiques et techniques.*

Denny, P., Luxton-Reilly, A., & Carpenter, D. (2014, June 21 - 25). Enhancing syntax error messages appears ineffectual. *ITiCSE'14*, pp. 273 - 278.

Denny, P., Luxton-Reily, A., Tempero, E., & Hendrickx, J. (2011, June 27 - 29). Understanding the syntax barrier for novices. *ITiCSE'11*, pp. 208 - 2012.

Frénay, B. (2017). Enseigner la programmation à l'université : challenges didactiques et pédagogiques. Dans J. Henry, A. Nguyen, & É. Vandeput, *L'informatique et le numérique dans la classe. Qui, quoi, comment?* Namur: Presse Universitaire de Namur.

Gomes, A., & Mendes, A. (2007a). An environment to improve programming education. *International Conference on Computer Systems and Technologies - CompSysTech'07*, (pp. IV.19-1 - IV.19-6).

Gomes, A., & Mendes, A. (2007b). Learning to program - difficulties and solutions. *International Conference on Engineering Education.* Coimbra, Portugal.

Hilburn, T. B., & Towhidnejad, M. (2000). Software Quality: A Curriculum Postscript? *ACM SIGCSE Bulletin, 32*(1), pp. 167 - 171.

Hughes, J., & Peiris, D. (2006). *ASSISTing CS1 Students to learn: Learning Approaches and Object-Oriented Programming.* Research Gate.

Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013). Why don't software developers use static analysis tools to find bugs? *Software Engineering (ICSE), 2013 35th International Conference on.* San Francisco.

Kasto, N., & Whalley, J. (2013). Measuring the difficulty of code comprehension tasks using software metrics. *Proceedings of the Fifteenth Australasian Computing Education Conference*, (pp. 59 - 65). Adelaide, Australia.

Kelleher, C., & Pausch, R. (2003). Lowering the Barriers to Programming: a survey of programming environments and languages for novice programmers. *ACM*.

Lahtinen, E., Ala-Mutka, K., & Järvinen, H.-M. (2005, June 27 - 29). A Study of the Difficulties of Novice Programmers. *ITiCSE*, pp. 14 - 18.

Lejeune, C. (2014). *Manuel d'analyse qualitative.* Louvain-La-Neuve: De Boeck Supérieur s.a.

Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010, November). The Scratch Programming Language And Environment. *ACM Transactions on Computing Education, 10*(4), p. Article 16.

McCracken, M., Kolikant, Y.-D., Almstrum, V., Laxer, C., Diaz, D., Thomas, L., . . . Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin, 33*, pp. 125 - 180.

Unamur - IHDCM031. (2015 - 2016). Part 5 - Software Quality.

Annexes

1. Exemple de fonction trop complexe

```
1   def place_ship(boards, player_id, ship, location, direction):
2       """
3   Specification OK
4       """
5
6       board1 = boards[0]
7       board2 = boards[1]
8
9       if ship == 'destroyer':
10          case_to_test=1
11          c = case_to_test
12      elif ship == 'submarine':
13          case_to_test=2
14          c = case_to_test
15      elif ship == 'battleship':
16          case_to_test=3
17          c = case_to_test
18      elif ship == 'aircraft carrier':
19          case_to_test=4
20          c = case_to_test
21      else:
22          raise ValueError('Unreal ship')
23
24      for i in board1 :
25          if location == board1[i]['case_name']:
26              if direction == 'up':
27                  i= int(i)-(10*c)
28                  if i not in board1 or board1[i]['case_name']== 'no_name':
29                      raise ValueError ('Impossible direction')
30                  else:
31                      if c == 1:
32                          test = test_case(board1, player_id, i)
33                          i= i+(10*c)
34                          test2 = test_case(board1, player_id, i)
35                          if test == False or test2 == False:
36                              raise ValueError ('New ship next to another one. Location must be changed')
37                          else:
38                              if player_id == 1:
39                                  board1[i-10]['statutP1'] = '#'
40                                  board1[i-10]['shipP1'] = 'd'
41                                  board1[i]['statutP1'] = '#'
42                                  board1[i]['shipP1'] = 'd'
43                              elif player_id == 2:
44                                  board1[i-10]['statutP2'] = '#'
45                                  board1[i-10]['shipP2'] = 'd'
46                                  board1[i]['statutP2'] = '#'
47                                  board1[i]['shipP2'] = 'd'
48                      elif c == 2:
```

```
                    test = test_case(board1, player_id, i)
                    i = i+(10*c)
                    test2 = test_case(board1, player_id, i)
                    if test == False or test2 == False:
                        raise ValueError ('New ship next to another one. Location must be changed')
                    else:
                        if player_id == 1:
                            board1[i-20]['statutP1'] = '#'
                            board1[i-20]['shipP1'] = 's'
                            board1[i-10]['statutP1'] = '#'
                            board1[i-10]['shipP1'] = 's'
                            board1[i]['statutP1'] = '#'
                            board1[i]['shipP1'] = 's'
                        elif player_id ==2:
                            board1[i-20]['statutP2'] = '#'
                            board1[i-20]['shipP2'] = 's'
                            board1[i-10]['statutP2'] = '#'
                            board1[i-10]['shipP2'] = 's'
                            board1[i]['statutP2'] = '#'
                            board1[i]['shipP2'] = 's'
                elif c == 3:
                    test = test_case(board1, player_id, i)
                    i = i+(10*c)
                    test2 = test_case(board1, player_id, i)
                    if test == False or test2 == False:
                        raise ValueError ('New ship next to another one. Location must be changed')
                    else:
                        if player_id == 1:
                            board1[i-30]['statutP1'] = '#'
                            board1[i-30]['shipP1'] = 'b'
                            board1[i-20]['statutP1'] = '#'
                            board1[i-20]['shipP1'] = 'b'
                            board1[i-10]['statutP1'] = '#'
                            board1[i-10]['shipP1'] = 'b'
                            board1[i]['statutP1'] = '#'
                            board1[i]['shipP1'] = 'b'
                        elif player_id == 2:
                            board1[i-30]['statutP2'] = '#'
                            board1[i-30]['shipP2'] = 'b'
                            board1[i-20]['statutP2'] = '#'
                            board1[i-20]['shipP2'] = 'b'
                            board1[i-10]['statutP2'] = '#'
                            board1[i-10]['shipP2'] = 'b'
                            board1[i]['statutP2'] = '#'
                            board1[i]['shipP2'] = 'b'
                elif c == 4:
                    test = test_case(board1, player_id, i)
                    i = i+(10*(c/2))
                    test2 = test_case(board1, player_id, i)
                    i = i+(10*(c/2))
                    test3 = test_case(board1, player_id, i)
```

```
100                 if test == False or test2 == False or test3 == False:
101                     raise ValueError ('New ship next to another one. Location must be changed')
102                 else:
103                     if player_id == 1:
104                         board1[i-40]['statutP1'] = '#'
105                         board1[i-40]['shipP1'] = 'a'
106                         board1[i-30]['statutP1'] = '#'
107                         board1[i-30]['shipP1'] = 'a'
108                         board1[i-20]['statutP1'] = '#'
109                         board1[i-20]['shipP1'] = 'a'
110                         board1[i-10]['statutP1'] = '#'
111                         board1[i-10]['shipP1'] = 'a'
112                         board1[i]['statutP1'] = '#'
113                         board1[i]['shipP1'] = 'a'
114                     elif player_id == 2:
115                         board1[i-40]['statutP2'] = '#'
116                         board1[i-40]['shipP2'] = 'a'
117                         board1[i-30]['statutP2'] = '#'
118                         board1[i-30]['shipP2'] = 'a'
119                         board1[i-20]['statutP2'] = '#'
120                         board1[i-20]['shipP2'] = 'a'
121                         board1[i-10]['statutP2'] = '#'
122                         board1[i-10]['shipP2'] = 'a'
123                         board1[i]['statutP2'] = '#'
124                         board1[i]['shipP2'] = 'a'
125             elif direction == 'down':
126                 i= int(i)+(10*c)
127                 if i not in board1 or board1[i]['case_name'] == 'no_name':
128                     raise ValueError ('Impossible direction')
129                 else:
130                     if c == 1:
131                         test = test_case(board1, player_id, i)
132                         i = i-(10*c)
133                         test2 = test_case(board1, player_id, i)
134                         if test == False or test2 == False:
135                             raise ValueError ('New ship next to another one. Location must be changed')
136                         else:
137                             if player_id == 1:
138                                 board1[i+10]['statutP1'] = '#'
139                                 board1[i+10]['shipP1'] = 'd'
140                                 board1[i]['statutP1'] = '#'
141                                 board1[i]['shipP1'] = 'd'
142                             elif player_id == 2:
143                                 board1[i+10]['statutP2'] = '#'
144                                 board1[i+10]['shipP2'] = 'd'
145                                 board1[i]['statutP2'] = '#'
146                                 board1[i]['shipP1'] = 'd'
147                     elif c ==2:
148                         test = test_case(board1, player_id, i)
149                         i = i-(10*c)
150                         test2 = test_case(board1, player_id, i)
```

```
151                     if test == False or test2 == False:
152                         raise ValueError ('New ship next to another one. Location must be changed')
153                     else:
154                         if player_id == 1:
155                             board1[i+20]['statutP1'] = '#'
156                             board1[i+20]['shipP1'] = 's'
157                             board1[i+10]['statutP1'] = '#'
158                             board1[i+10]['shipP1'] = 's'
159                             board1[i]['statutP1'] = '#'
160                             board1[i]['shipP1'] = 's'
161                         elif player_id ==2:
162                             board1[i+20]['statutP2'] = '#'
163                             board1[i+20]['shipP2'] = 's'
164                             board1[i+10]['statutP2'] = '#'
165                             board1[i+10]['shipP2'] = 's'
166                             board1[i]['statutP2'] = '#'
167                             board1[i]['shipP2'] = 's'
168                 elif c == 3:
169                     test = test_case(board1, player_id, i)
170                     i = i-(10*c)
171                     test2 = test_case(board1, player_id, i)
172                     if test == False or test2 == False:
173                         raise ValueError ('New ship next to another one. Location must be changed')
174                     else:
175                         if player_id == 1:
176                             board1[i+30]['statutP1'] = '#'
177                             board1[i+30]['shipP1'] = 'b'
177                             board1[i+20]['statutP1'] = '#'
178                             board1[i+20]['shipP1'] = 'b'
179                             board1[i+10]['statutP1'] = '#'
180                             board1[i+10]['shipP1'] = 'b'
181                             board1[i]['statutP1'] = '#'
182                             board1[i]['shipP1'] = 'b'
183                         elif player_id == 2:
184                             board1[i+30]['statutP2'] = '#'
185                             board1[i+30]['shipP2'] = 'b'
186                             board1[i+20]['statutP2'] = '#'
187                             board1[i+20]['shipP2'] = 'b'
188                             board1[i+10]['statutP2'] = '#'
189                             board1[i+10]['shipP2'] = 'b'
190                             board1[i]['statutP2'] = '#'
191                             board1[i]['shipP2'] = 'b'
192                 elif c == 4:
193                     test = test_case(board1, player_id, i)
194                     i = i-(10*(c/2))
195                     test2 = test_case(board1, player_id, i)
196                     i = i-(10*(c/2))
197                     test3 = test_case(board1, player_id, i)
198                     if test == False or test2 == False or test3 == False:
199                         raise ValueError ('New ship next to another one. Location must be changed')
200                     else:
```

```
                    if player_id == 1:
                        board1[i+40]['statutP1'] = '#'
                        board1[i+40]['shipP1'] = 'a'
                        board1[i+30]['statutP1'] = '#'
                        board1[i+30]['shipP1'] = 'a'
                        board1[i+20]['statutP1'] = '#'
                        board1[i+20]['shipP1'] = 'a'
                        board1[i+10]['statutP1'] = '#'
                        board1[i+10]['shipP1'] = 'a'
                        board1[i]['statutP1'] = '#'
                        board1[i]['shipP1'] = 'a'
                    elif player_id == 2:
                        board1[i+40]['statutP2'] = '#'
                        board1[i+40]['shipP2'] = 'a'
                        board1[i+30]['statutP2'] = '#'
                        board1[i+30]['shipP2'] = 'a'
                        board1[i+20]['statutP2'] = '#'
                        board1[i+20]['shipP2'] = 'a'
                        board1[i+10]['statutP2'] = '#'
                        board1[i+10]['shipP2'] = 'a'
                        board1[i]['statutP2'] = '#'
                        board1[i]['shipP2'] = 'a'
            elif direction == 'right':
                i = i+(1*c)
                if i not in board1 or board1[i]['case_name'] == 'no_name':
                    raise ValueError ('Impossible direction')
                else:
                    if c == 1:
                        test = test_case(board1, player_id, i)
                        i = i-(1*c)
                        test2 = test_case(board1, player_id, i)
                        if test == False or test2 == False:
                            raise ValueError ('New ship next to another one. Location must be changed')
                        else:
                            if player_id == 1:
                                board1[i+1]['statutP1'] = '#'
                                board1[i+1]['shipP1'] = 'd'
                                board1[i]['statutP1'] = '#'
                                board1[i]['shipP1'] = 'd'
                            elif player_id == 2:
                                board1[i+1]['statutP2'] = '#'
                                board1[i+1]['shipP2'] = 'd'
                                board1[i]['statutP2'] = '#'
                                board1[i]['shipP2'] = 'd'
                    elif c ==2:
                        test = test_case(board1, player_id, i)
                        i = i-(1*c)
                        test2 = test_case(board1, player_id, i)
                        if test == False or test2 == False:
                            raise ValueError ('New ship next to another one. Location must be changed')
                        else:
```

```
252                     if player_id == 1:
253                         board1[i+2]['statutP1'] = '#'
254                         board1[i+2]['shipP1'] = 's'
255                         board1[i+1]['statutP1'] = '#'
256                         board1[i+1]['shipP1'] = 's'
257                         board1[i]['statutP1'] = '#'
258                         board1[i]['shipP1'] = 's'
259                     elif player_id ==2:
260                         board1[i+2]['statutP2'] = '#'
261                         board1[i+2]['shipP2'] = 's'
262                         board1[i+1]['statutP2'] = '#'
263                         board1[i+1]['shipP2'] = 's'
264                         board1[i]['statutP2'] = '#'
265                         board1[i]['shipP2'] = 's'
266                 elif c == 3:
267                     test = test_case(board1, player_id, i)
268                     i = i-(1*c)
269                     test2 = test_case(board1, player_id, i)
270                     if test == False or test2 == False:
271                         raise ValueError ('New ship next to another one. Location must be changed')
272                     else:
273                         if player_id == 1:
274                             board1[i+3]['statutP1'] = '#'
275                             board1[i+3]['shipP1'] = 'b'
276                             board1[i+2]['statutP1'] = '#'
277                             board1[i+2]['shipP1'] = 'b'
278                             board1[i+1]['statutP1'] = '#'
279                             board1[i+1]['shipP1'] = 'b'
280                             board1[i]['statutP1'] = '#'
281                             board1[i]['shipP1'] = 'b'
282                         elif player_id == 2:
283                             board1[i+3]['statutP2'] = '#'
284                             board1[i+3]['shipP2'] = 'b'
285                             board1[i+2]['statutP2'] = '#'
286                             board1[i+2]['shipP2'] = 'b'
287                             board1[i+1]['statutP2'] = '#'
288                             board1[i+1]['shipP2'] = 'b'
289                             board1[i]['statutP2'] = '#'
290                             board1[i]['shipP2'] = 'b'
291                 elif c == 4:
292                     test = test_case(board1, player_id, i)
293                     i = i-(1*(c/2))
294                     test2 = test_case(board1, player_id, i)
295                     i = i-(1*(c/2))
296                     test3 = test_case(board1, player_id, i)
297                     if test == False or test2 == False or test3 == False:
298                         raise ValueError ('New ship next to another one. Location must be changed')
299                     else:
300                         if player_id == 1:
301                             board1[i+4]['statutP1'] = '#'
302                             board1[i+4]['shipP1'] = 'a'
```

```
                        board1[i+3]['statutP1'] = '#'
                        board1[i+3]['shipP1'] = 'a'
                        board1[i+2]['statutP1'] = '#'
                        board1[i+2]['shipP1'] = 'a'
                        board1[i+1]['statutP1'] = '#'
                        board1[i+1]['shipP1'] = 'a'
                        board1[i]['statutP1'] = '#'
                        board1[i]['shipP1'] = 'a'
                    elif player_id == 2:
                        board1[i+4]['statutP2'] = '#'
                        board1[i+4]['shipP2'] = 'a'
                        board1[i+3]['statutP2'] = '#'
                        board1[i+3]['shipP2'] = 'a'
                        board1[i+2]['statutP2'] = '#'
                        board1[i+2]['shipP2'] = 'a'
                        board1[i+1]['statutP2'] = '#'
                        board1[i+1]['shipP2'] = 'a'
                        board1[i]['statutP2'] = '#'
                        board1[i]['shipP2'] = 'a'
        elif direction == 'left':
            i= int(i)-(1*c)
            if i not in board1 or board1[i]['case_name']== 'no_name':
                raise ValueError ('Impossible direction')
            else:
                if c == 1:
                    test = test_case(board1, player_id, i)
                    i = i+(1*c)
                    test2 = test_case(board1, player_id, i)
                    if test == False or test2 == False:
                        raise ValueError ('New ship next to another one. Location must be changed')
                    else:
                        if player_id == 1:
                            board1[i-1]['statutP1'] = '#'
                            board1[i-1]['shipP1'] = 'd'
                            board1[i]['statutP1'] = '#'
                            board1[i]['shipP1'] = 'd'
                        elif player_id == 2:
                            board1[i-1]['statutP2'] = '#'
                            board1[i-1]['shipP2'] = 'd'
                            board1[i]['statutP2'] = '#'
                            board1[i]['shipP2'] = 'd'
                elif c ==2:
                    test = test_case(board1, player_id, i)
                    i = i+(1*c)
                    test2 = test_case(board1, player_id, i)
                    if test == False or test2 == False:
                        raise ValueError ('New ship next to another one. Location must be changed')
                    else:
                        if player_id == 1:
                            board1[i-2]['statutP1'] = '#'
                            board1[i-2]['shipP1'] = 's'
```

```
                    board1[i-1]['statutP1'] = '#'
                    board1[i-1]['shipP1'] = 's'
                    board1[i]['statutP1'] = '#'
                    board1[i]['shipP1'] = 's'
                elif player_id ==2:
                    board1[i-2]['statutP2'] = '#'
                    board1[i-2]['shipP2'] = 's'
                    board1[i-1]['statutP2'] = '#'
                    board1[i-1]['shipP2'] = 's'
                    board1[i]['statutP2'] = '#'
                    board1[i]['shipP2'] = 's'
            elif c == 3:
                test = test_case(board1, player_id, i)
                i = i+(1*c)
                test2 = test_case(board1, player_id, i)
                if test == False or test2 == False:
                    raise ValueError ('New ship next to another one. Location must be changed')
                else:
                    if player_id == 1:
                        board1[i-3]['statutP1'] = '#'
                        board1[i-3]['shipP1'] = 'b'
                        board1[i-2]['statutP1'] = '#'
                        board1[i-2]['shipP1'] = 'b'
                        board1[i-1]['statutP1'] = '#'
                        board1[i-1]['shipP1'] = 'b'
                        board1[i]['statutP1'] = '#'
                        board1[i]['shipP1'] = 'b'
                    elif player_id == 2:
                        board1[i-3]['statutP2'] = '#'
                        board1[i-3]['shipP2'] = 'b'
                        board1[i-2]['statutP2'] = '#'
                        board1[i-2]['shipP2'] = 'b'
                        board1[i-1]['statutP2'] = '#'
                        board1[i-1]['shipP2'] = 'b'
                        board1[i]['statutP2'] = '#'
                        board1[i]['shipP2'] = 'b'
            elif c == 4:
                test = test_case(board1, player_id, i)
                i = i+(1*(c/2))
                test2 = test_case(board1, player_id, i)
                i = i+(1*(c/2))
                test3 = test_case(board1, player_id, i)
                if test == False or test2 == False or test3 == False:
                    raise ValueError ('New ship next to another one. Location must be changed')
                else:
                    if player_id == 1:
                        board1[i-4]['statutP1'] = '#'
                        board1[i-4]['shipP1'] = 'a'
                        board1[i-3]['statutP1'] = '#'
                        board1[i-3]['shipP1'] = 'a'
                        board1[i-2]['statutP1'] = '#'
```

```
405                         board1[i-2]['shipP1'] = 'a'
406                         board1[i-1]['statutP1'] = '#'
407                         board1[i-1]['shipP1'] = 'a'
408                         board1[i]['statutP1'] = '#'
409                         board1[i]['shipP1'] = 'a'
410                     elif player_id == 2:
411                         board1[i-4]['statutP2'] = '#'
412                         board1[i-4]['shipP2'] = 'a'
413                         board1[i-3]['statutP2'] = '#'
414                         board1[i-3]['shipP2'] = 'a'
415                         board1[i-2]['statutP2'] = '#'
416                         board1[i-2]['shipP2'] = 'a'
417                         board1[i-1]['statutP2'] = '#'
418                         board1[i-1]['shipP2'] = 'a'
419                         board1[i]['statutP2'] = '#'
420                         board1[i]['shipP2'] = 'a'
421             else:
422                 raise ValueError('wrong direction')
423     boards = (board1, board2)
424     see(boards, player_id)
425
426     #save
427     fh = open('battleship.pkl', 'wb')
428     pickle.dump(boards, fh)
429     fh.close
430     print 'Game saved.'
```

2. Liste des règles Python implémentées par SonarQube

- "<>" should not be used to test inequality
- "\" should only be used as an escape character outside of raw strings
- "__exit__" should accept type, value, and traceback arguments
- "__init__" should not return a value
- "break" and "continue" should not be used outside a loop
- "FIXME" tags should be handled
- "pass" should not be used needlessly
- "return" and "yield" cannot be used in the same function
- "yield" and "return" should not be used outside functions
- __exit__ must accept 3 arguments: type, value, traceback
- __future__ import is not the first non docstring statement
- __init__ method from a non direct base class is called
- __init__ method from base class is not called
- __init__ method is a generator
- __iter__ returns non-iterator
- A field should not duplicate the name of its containing class
- Abstract class not referenced
- Abstract class used too few times
- Abstract method is not overridden
- Access of nonexistent member
- Access to a protected member of a client class
- Access to member before its definition
- Accessing nonexistent member (type information incomplete)
- Analysis failed
- Anomalous backslash escape
- Anomalous Unicode escape in byte string
- Arguments number discrepancy
- Assert called on a 2-uple
- Assigning to function call which doesn't return
- Assigning to function call which only returns None
- Attempting to unpack a non-sequence
- Attribute defined outside __init__
- Avoid catching an exception which doesn't inherit from BaseException
- Backticks should not be used
- Bad continuation
- Bad except clauses order
- Bad first argument given to super
- Bad indentation
- Bad option value
- Black listed name

- Branches should have sufficient coverage by unit tests
- Calling of not callable
- Catching too general exception
- Class has no __init__ method
- Class method should have "cls" as first argument
- Class names should comply with a naming convention
- Classes should not be too complex
- Collapsible "if" statements should be merged
- Comma not followed by a space
- Comments should not be located at the end of lines of code
- Conditions in related "if/elif/else if" statements should not have the same condition
- Control flow statements "if", "for", "while", "try" and "with" should not be nested too deeply
- Cyclic import
- Dangerous default value as argument
- Docstrings should be defined
- Duplicate argument name in function definition
- Duplicate key in dictionary
- Duplicate keyword argument in function call
- Else clause on loop without a break statement
- Empty docstring
- Error while code parsing
- Except doesn't do anything
- Exception doesn't inherit from standard "Exception" class
- Exception to catch is the result of a binary operation
- Expected mapping for format string
- Explicit return in __init__
- Expression is assigned to nothing
- Failed to resolve interfaces
- Failed unit tests should be fixed
- Field names should comply with a naming convention
- Files should contain an empty new line at the end
- Files should not be too complex
- Files should not have too many lines
- Final newline missing
- Format detection error
- Format string dictionary key should be a string
- Format string ends in middle of conversion specifier
- Function names should comply with a naming convention
- Functions should not be too complex

- Functions should not contain too many return statements
- Functions, methods and lambdas should not have too many parameters
- Global variable undefined at the module level
- Ignored builtin module
- Ignoring entire file
- Implemented interface must be a class
- Implicit unpacking of exceptions is not supported in Python 3
- Interface not implemented
- Internal Pylint error
- Invalid __slots__ object
- Invalid mode for open
- Invalid name
- Invalid object in __all__, must contain only strings
- Invalid object in __slots__, must contain only non empty strings
- IP addresses should not be hardcoded
- Jump statements should not be followed by other statements
- Lambda may not be necessary
- Line too long
- Lines should have sufficient coverage by unit tests
- Lines should not be too long
- Lines should not end with trailing whitespaces
- Local variable and function parameter names should comply with a naming convention
- Locally disabling message
- Locally enabling message
- Logging format string ends in middle of conversion specifier
- Long suffix "L" should be upper case
- map/filter on lambda could be replaced by comprehension
- Metaclass class method first argument
- Metaclass method should have "mcs" as first argument
- Method could be a function
- Method has no argument
- Method hidden by attribute of super class
- Method names should comply with a naming convention
- Method should have "self" as first argument
- Methods and field names should not differ only by capitalization
- Methods that don't access instance data should be "static"
- Missing argument to reversed()
- Missing argument to super()
- Missing docstring
- Missing key in format string dictionary

- Missing method from interface
- Missing required attribute
- Mixed tabs/spaces indentation
- Mixing named and unnamed conversion specifiers in format string
- Module imports itself
- Module names should comply with a naming convention
- More than one statement on a single line
- Multiple values passed for parameter in function call
- Nested blocks of code should not be left empty
- New-style classes should be used
- No exception type(s) specified
- Non-ASCII characters found but no encoding specified (PEP 263)
- Not enough arguments for format string
- Not enough arguments for logging format string
- NotImplemented raised - should raise NotImplementedError
- Old-style class defined
- Operator not followed by a space
- Operator not preceded by a space
- Parentheses should not be used after certain keywords
- Parser failure
- Passing unexpected keyword argument in function call
- Possible unbalanced tuple unpacking
- Pre-increment and pre-decrement should not be used
- Raising a new style class which doesn't inherit from BaseException
- Raising a string exception
- Raising only allowed for classes, instances or strings
- Redefined function/class/method
- Redefining built-in
- Redefining name from outer scope
- Redefining name in exception handler
- Regular expression on comment
- Reimport
- Relative import
- Return outside function
- Return with argument inside generator
- Sections of code should not be "commented out"
- Signature discrepancy
- Similar lines
- Skipped unit tests should be either removed or fixed
- Source files should have a sufficient density of comment lines
- Source files should not have any duplicated blocks

- Source line cannot be decoded using the specified source file encoding
- Specify string format arguments as logging function parameters
- Statement in finally block may swallow exception
- Statement seems to have no effect
- Statements should be on separate lines
- Static method with "self" or "cls" as first argument
- String statement has no effect
- Suspicious argument in lstrip/rstrip
- Syntax error
- Task marker found
- The "exec" statement should not be used
- The "print" statement should not be used
- The first reversed() argument is not a sequence
- TODO and FIXME comments should contain a reference to a person
- Too few arguments
- Too few public methods
- Too many ancestors
- Too many arguments
- Too many arguments for format string
- Too many arguments for logging format string
- Too many branches
- Too many instance attributes
- Too many lines in module
- Too many local variables
- Too many positional arguments for function call
- Too many public methods
- Too many return statements
- Too many statements
- Trailing whitespace
- Two branches in the same conditional structure should not have exactly the same implementation
- Unable to check methods signature
- Unable to consider inline option
- Unable to import module
- Unable to run raw checkers on built-in module
- Unassigned global variable
- Undefined name
- Undefined variable
- Undefined variable name in __all__
- Unexpected inferred value
- Unknown encoding specified

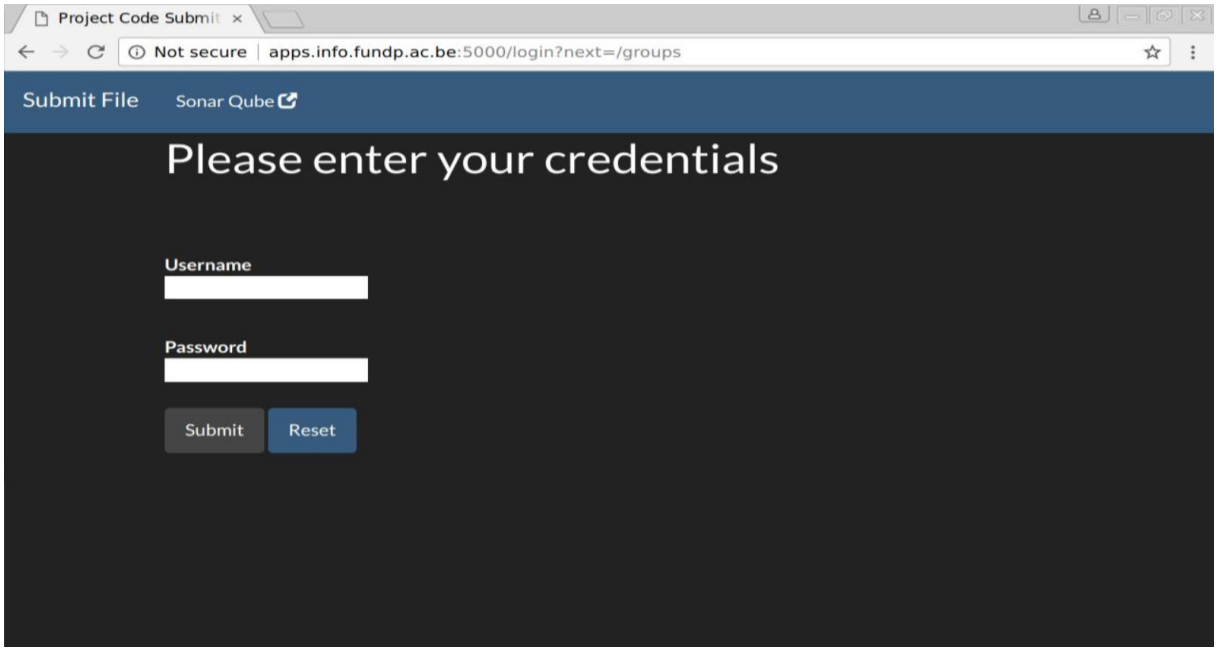
- Unnecessary parentheses
- Unnecessary pass statement
- Unnecessary semicolon
- Unreachable code
- Unrecognized file option
- Unsupported format character
- Unsupported logging format character
- Unused argument
- Unused import
- Unused import from wildcard import
- Unused key in format string dictionary
- Unused variable
- Usage of 'break' or 'continue' outside of a loop
- Use l as long integer identifier
- Use of "property" on an old style class
- Use of __slots__ on an old style class
- Use of a non-existent operator
- Use of eval
- Use of super on an old style class
- Use of the <> operator
- Use of the `` operator
- Use of the exec statement
- Use raise ErrorClass(args) instead of raise ErrorClass, args.
- Used * or ** magic
- Used black listed builtin function
- Useless parentheses around expressions should be removed to prevent any misunderstanding
- Uses of a deprecated module
- Using possibly undefined loop variable
- Using the global statement
- Using the global statement at the module level
- Using variable before assignment
- Wildcard import
- Wrong encoding specified
- Wrong number of spaces around an operator, bracket, or comma, or before a block opener colon
- XPath rule
- Yield outside function

3. Guide d'utilisation de Code Submitter

How to run the tool

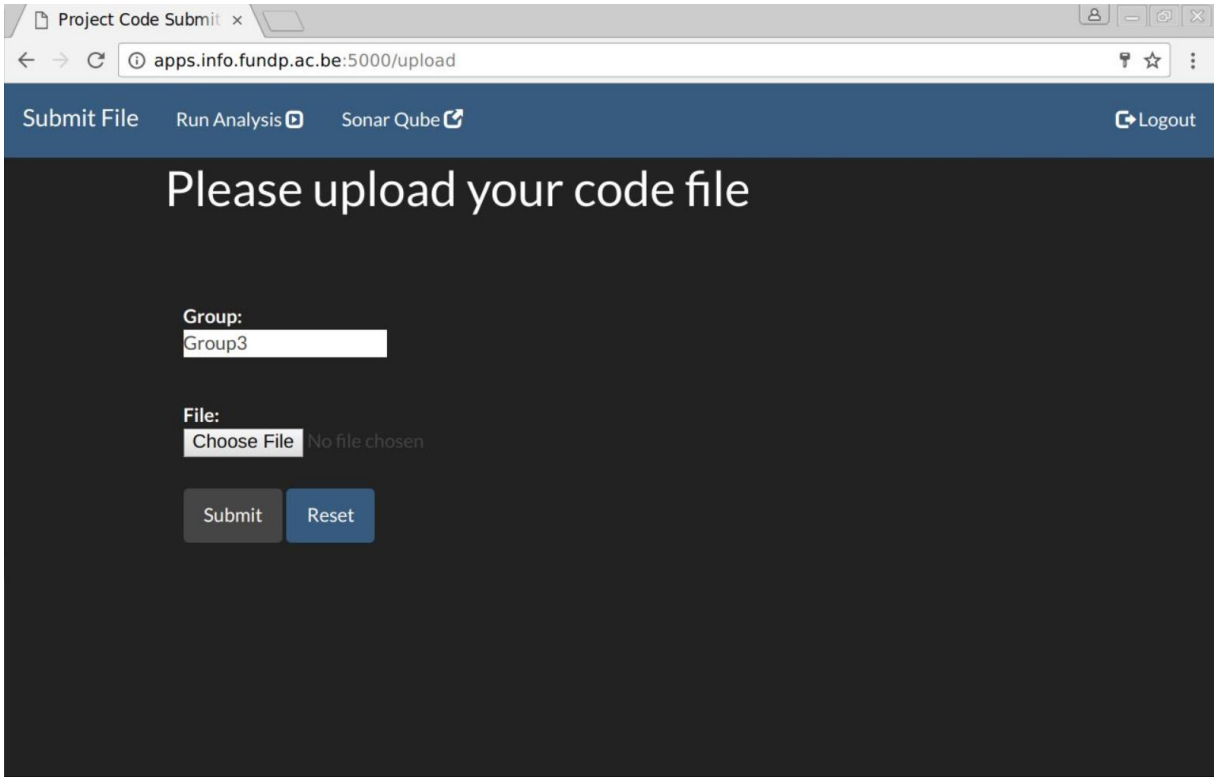
1) *Log into « code submitter » :*

Access the url apps.info.fundp.ac.be:5000

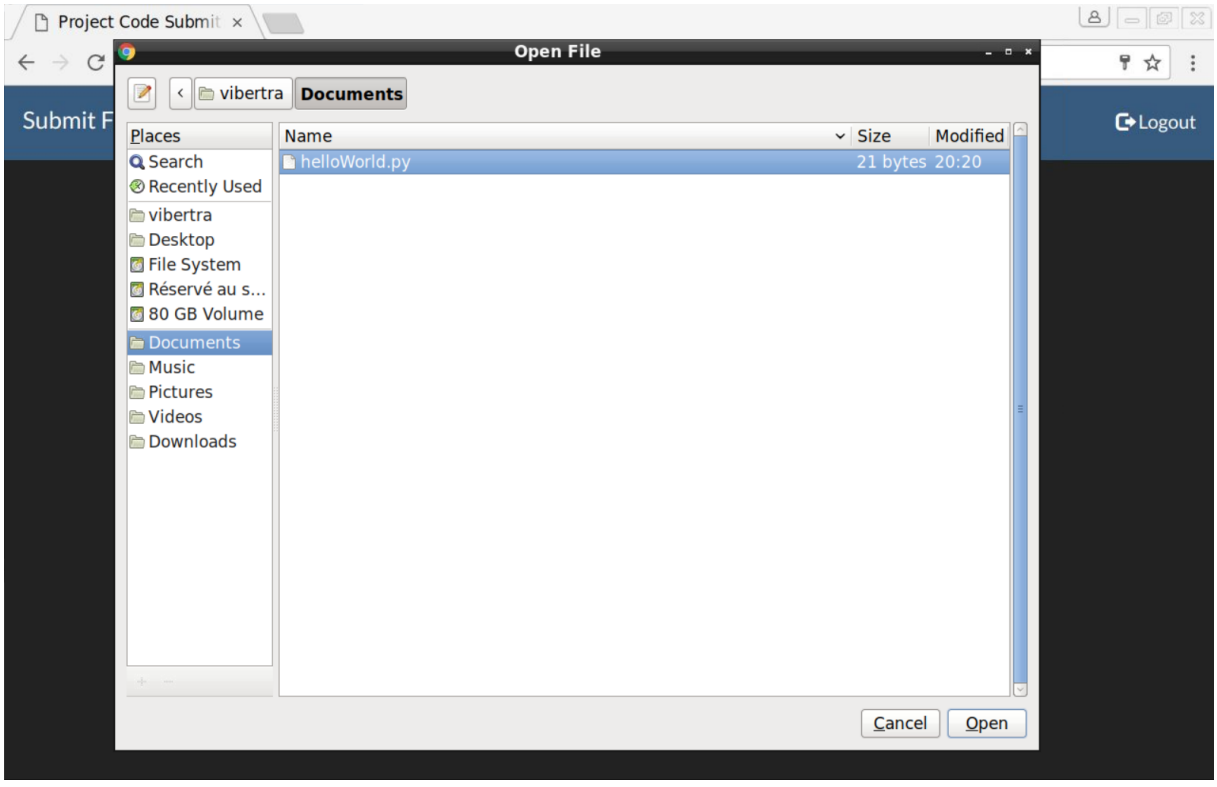


Enter your credentials. Please note that the username and the login are case sensitives : Group1 is different than group1

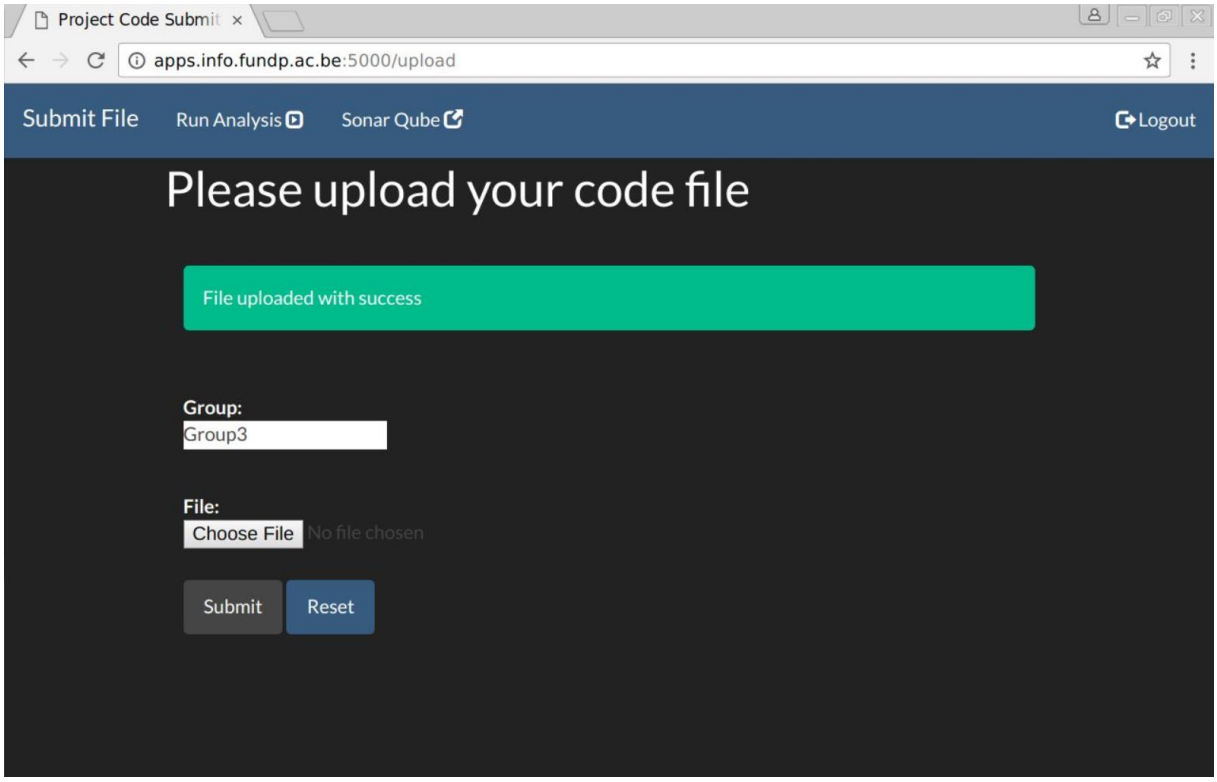
2) Upload a file :



Click on the « Choose File » button, select your python file (only file with .py).

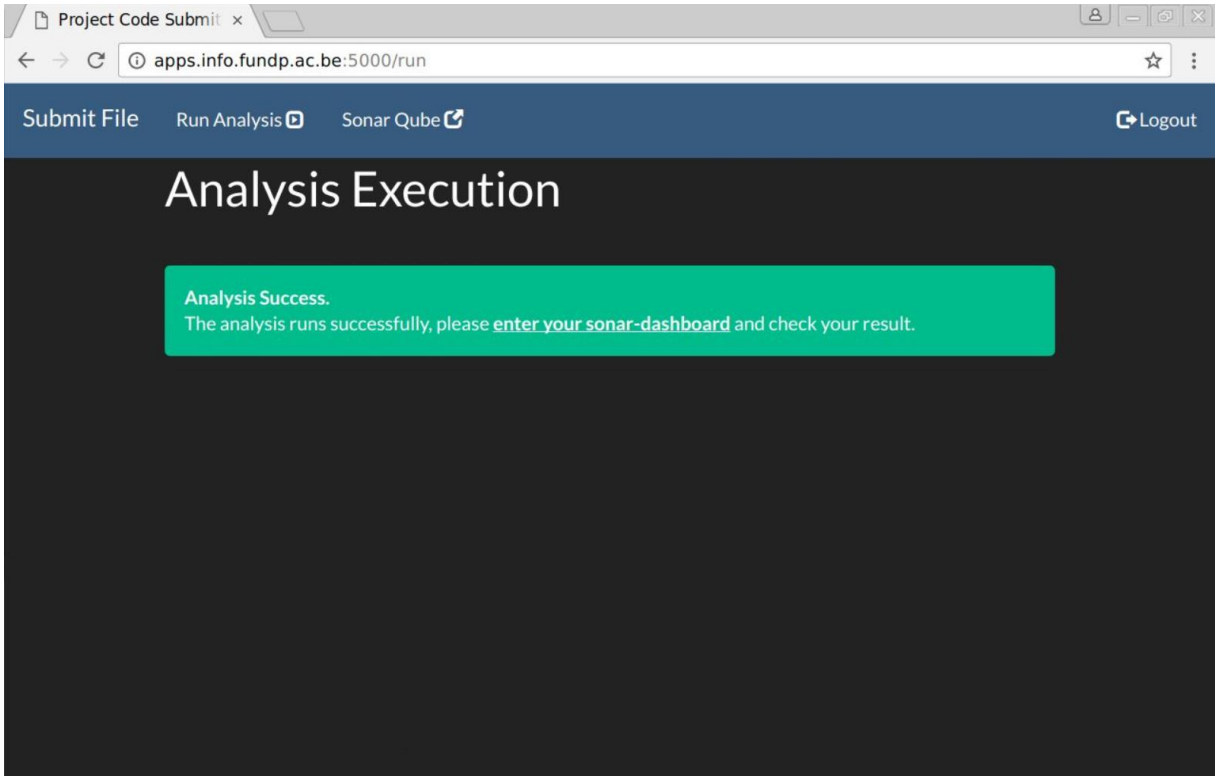


Once the file is selected, click the « Submit » button :



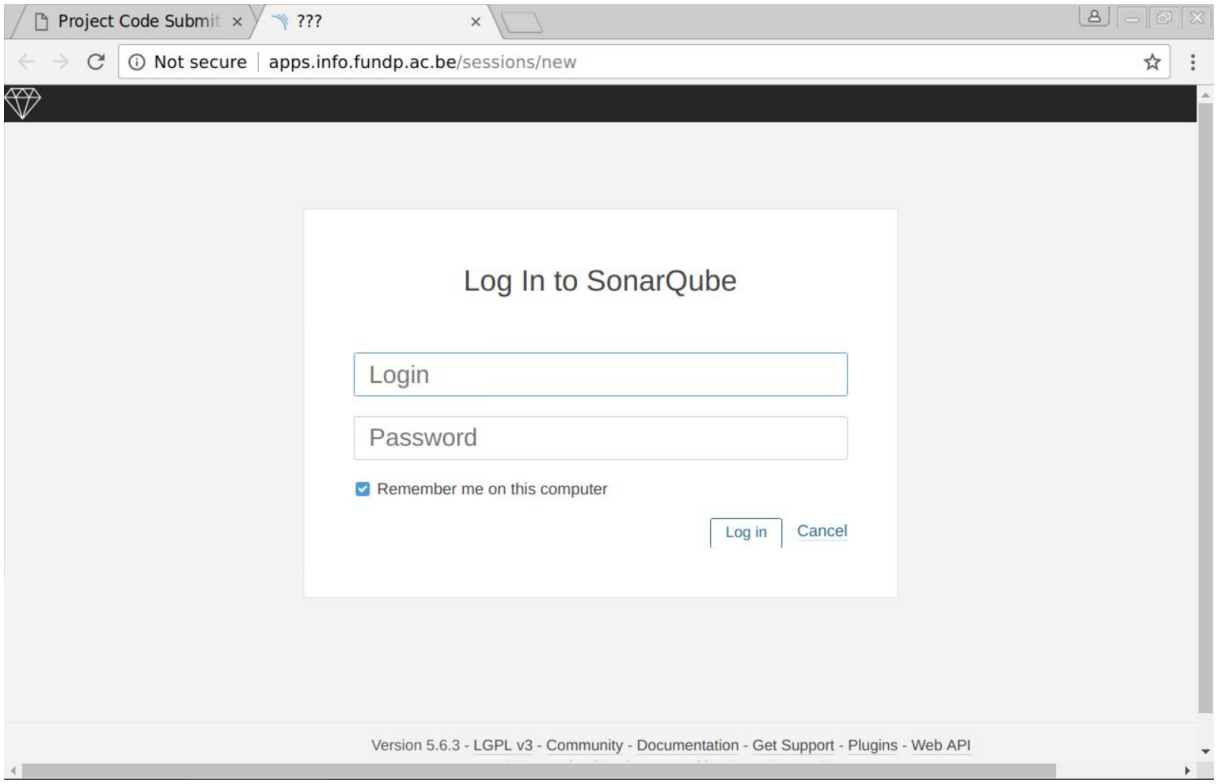
3) *Run the analysis*

Click the menu item « Run Analysis » and wait the page to be loaded (it can take few seconds):



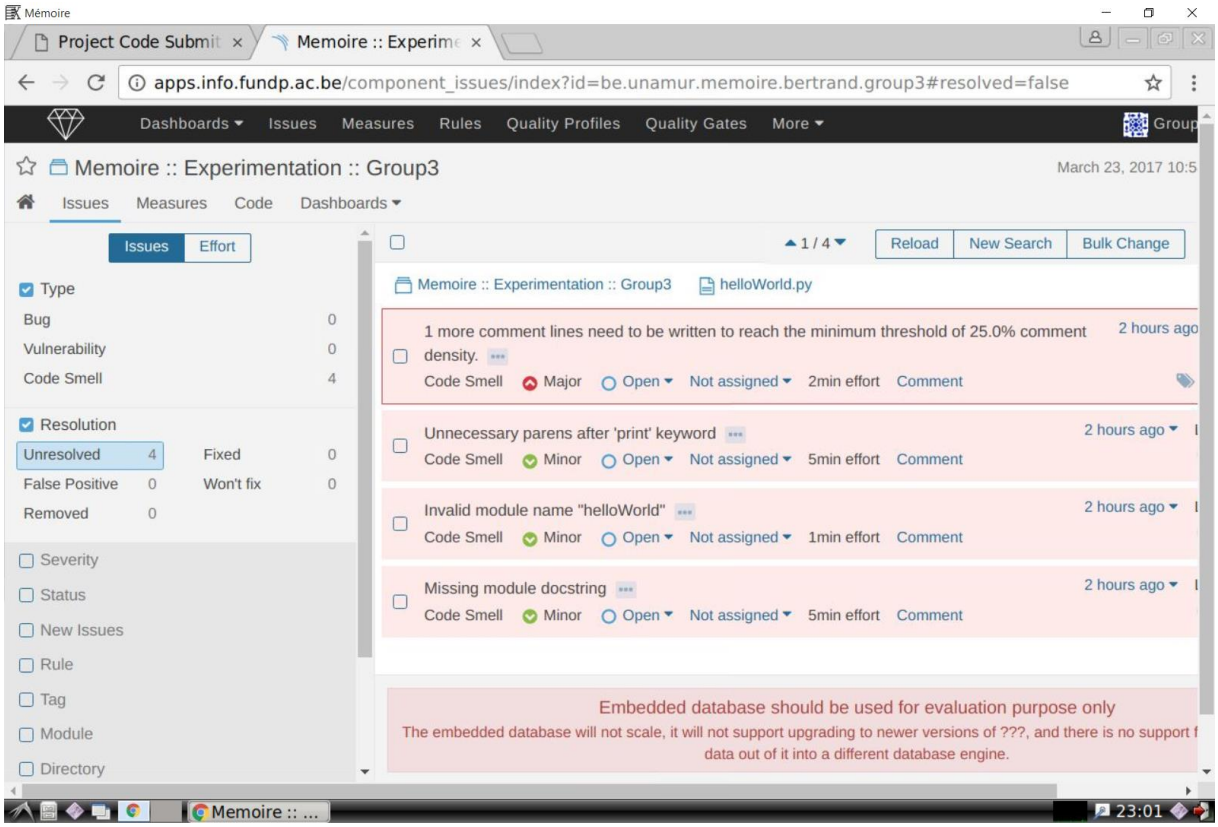
4) *Open the Sonar Qube Dashboard :*

Click the menu item « Sonar Qube » or the link present in the success message :



Enter the same credentials as before

5) *Use the Dashboard to improve your code :*



Enjoy !